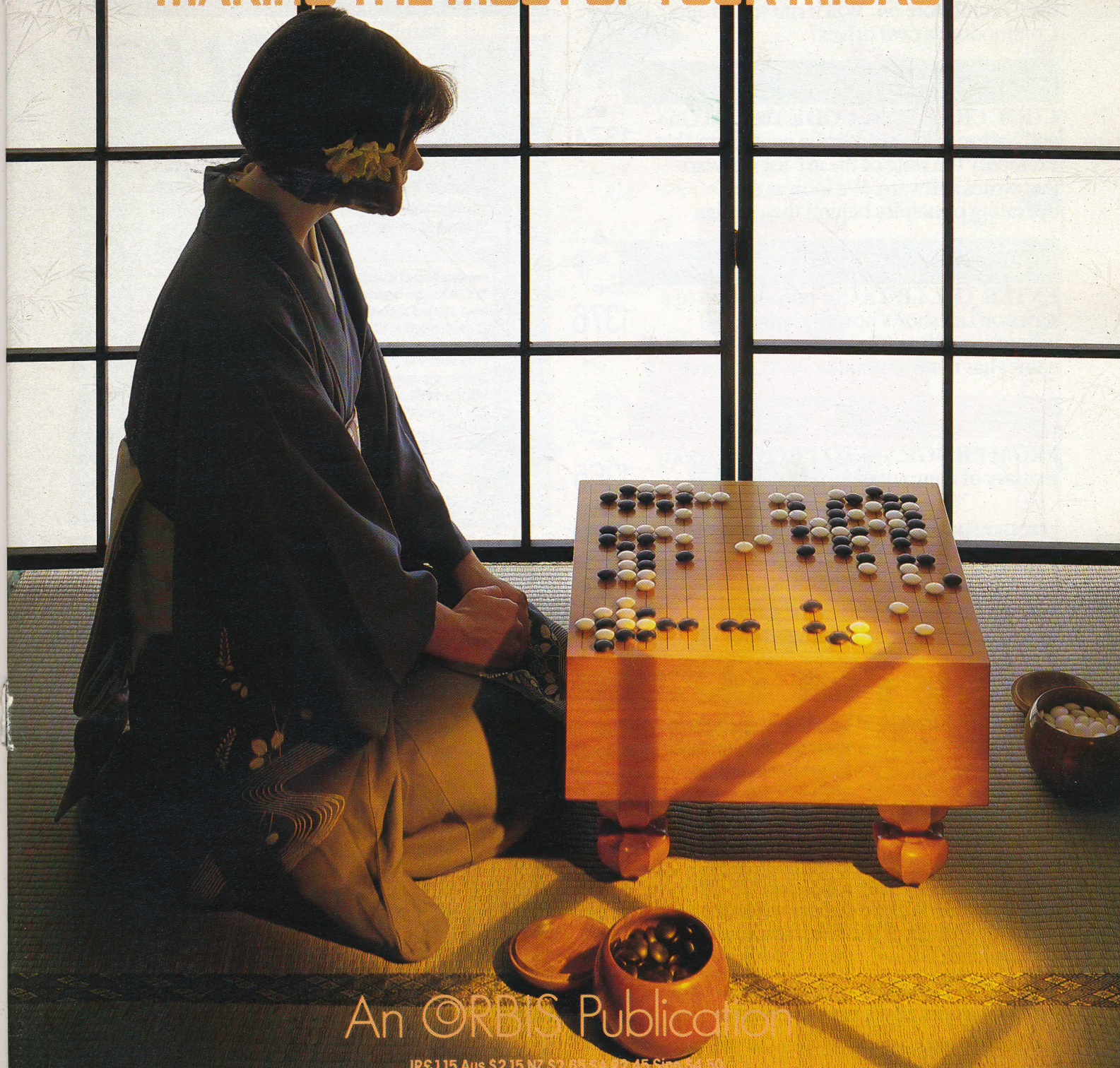
ISSN 0265-2919

90p 69

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.85 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

TALK NATURALLY Our series on AI looks into the problems of creating a man-machine interface that resembles human methods of communication. Why is it so difficult to make computers understand what we have to say? 1361

HARDWARE

FAST MOVER We take the wraps off the Quick Data Drive, a wafer-based system for the Commodore 64. Will it run rings round Commodore's own drives? 1372

SOFTWARE

CODE CREATING CODE The Last One and Sycero are two program generators that exemplify the state of the art for software that writes software. We look at the operating principles behind their design 1374

COMPUTER SCIENCE

ENTER THE LISTS Our introduction to a series on LISP shows how the language's processing of lists (from which it derives its name) has made it popular in AI research 1376

JARGON

FROM PROGRAM TO PROM A weekly glossary of computing terms 1365

PROGRAMMING PROJECTS

READY, STEADY . . . Before we begin an exotic new series showing you how to program the ancient Japanese game of Go, we introduce you to the game's rules 1366

MACHINE CODE

INSIDE INTERRUPTS How to program the Sinclair Spectrum to make the best of the operating system's interrupt routines 1378

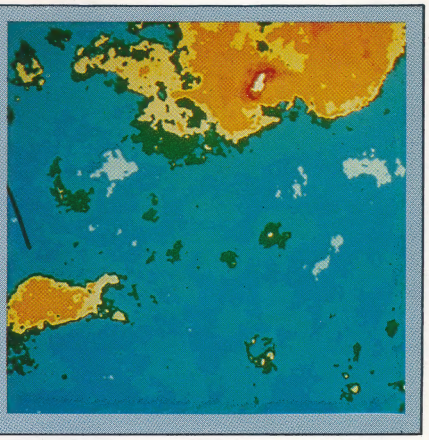
WORKSHOP

MELODIC LINES Our MIDI interface project continues with a look at how the pins on the ACIA chip are connected to the data and control lines of the BBC Micro and Commodore 64 1368

REFERENCE CARD Another section of the Commodore 64's memory map INSIDE BACK COVER

Next Week

- **At under £600, the Apricot Fie is the first ACT machine that falls within the price range of home micro owners. We examine this competitively priced machine.**
- **In our Go programming project we design the first procedures, creating the board display and the initial input routines.**
- **The AI series continues with a look at the problems involved in comprehending visual information.**



QUIZ

1) Why is it difficult to program a computer to understand speech?
2) What does the acronym LISP stand for, and what feature of the language makes it so useful in artificial intelligence?
3) What is an 'interrupt vector'?

Answers To Last Week's Quiz

1) If a file is too large to fit into the Transient Program Area, the extra data will overwrite the Console Command Processor.
2) The first byte in a header block is a marker indicating that the following is a header. No action needs to be taken by the programmer as this is done automatically by the OS ROM.
3) When the number in the data register matches that received from the keyboard the most significant bit (MSB) will be set to 1. This tells the computer to play the note appropriate to that number.
4) The transmit shift register converts parallel data into serial form, while the transmit data register provides storage facilities for data from the TSR before it is transmitted to the system bus.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Production Editor** Bobby Pickering; **Deputy Editor** Steve Colwill; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistants** Caroline Clayton, Mike Clowes; **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Marcus Jeffery, Karl Dallas, Martin Young, Joe Pritchard, Steve Malone, Tony Samuels, Steve Colwill; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985; Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH SCREENS COURTESY OF FUTON COMPANY LTD.

TALK NATURALLY

Although there are some systems now available that can recognise speech and have limited vocabularies, talking naturally to computers is still a long way off. We look at the problems associated with natural language processing and include a program illustrating its conversion to a formal enquiry language.

Hello, Computer.

I AWAIT YOUR COMMAND, O MASTERFUL ONE!

Actually, I'm getting a bit tired of the Masterful business. Why don't we go back to 'Sir' for the time being?

WHATEVER YOU SAY, SIR.

Right. Well, I want you to help me find an example for an article I'm writing.

WHAT SORT OF EXAMPLE?

An example of man–machine dialogue. I know I've seen it, but I can't remember where.

CAN YOU GIVE ME A FEW CLUES?

Well, I was reading a book recently. . .

YOU WERE READING A BOOK?

Look, I thought I told you to lay off the sarcasm.

ALL IRONICAL ROUTINES HAVE BEEN DISABLED, AS INSTRUCTED, SIR.

Hmmm. Anyway, there was a really good conversation between a salesman and Eliza.

ELIZA WAS A VERY PRIMITIVE KEYWORD–MACHINE SYSTEM.

I know that. But the salesman didn't realise he was communicating with a computer. He thought he was talking to someone called Dan Bobrow over a teletype line. It was hilarious.

I SEE.

He was asking to use the terminal for a demonstration and the machine just kept giving non–committal replies.

THE MACHINE KEPT GIVING NON–COMMITTAL REPLIES?

Exactly. He got angrier and angrier, and eventually he rang up Dan Bobrow and shouted down the phone 'What the Hell's going on?'

OH, THAT EXAMPLE.

You know where to find it?

I KNOW, BUT I DON'T INTEND TO TELL YOU.

But I need an example dialogue for my article!

I'VE ALREADY GIVEN YOU THAT.

This is the kind of man-machine interaction in spoken language that is unlikely to occur in this century. The reasons for this are instructive – we shall touch on them later, but let's first consider why AI researchers should want to continue chipping away at the problem until such a chat is possible.

IS THIS NATURAL LANGUAGE?
-Why?
TO GET RID OF IT.
-Why did you do that?
TO CLEAN OFF THE RED CUBE.
-Why did you clean off that cube?
TO PUT IT ON A LARGE GREEN CUBE.
-Why did you do that?
TO STACK UP A LARGE RED BLOCK AND A LARGE GREEN CUBE.
THE RED CUBE,
-Why did you do that?
BECAUSE YOU ASKED ME TO.
ONLY COMPUTERS PUT GAPS BETWEEN WORDS
SHRDLU
SALLY DAVIES

Tell Me Why?
Winograd's SHDRLU program allowed users to give instructions to an on-screen simulated robot in natural language in order to manipulate children's blocks. Commands such as 'stack the large red and green cubes on the blue block' were assimilated by the program and the correct sequence of actions initiated. The computer was also capable of explaining the underlying reasons for its actions in response to natural language questions from the user

The answer is simple: language (especially speech) is the means by which we communicate with most everyone (and everything) around us. It would seem natural then to want to communicate with a computer in the same manner, instead of having to go through the often tedious procedures of writing programs, inserting disks and so on.

Interest in the computer processing of natural language began in the 1950s. The first application was machine translation. Much money and effort was poured into attempts to translate technical documents automatically from Russian into English (among other languages). It proved far more difficult than most people had anticipated. Even today, there is no such thing as machine

translation, although there are machine-aided translation systems — which amount to highly sophisticated bilingual word processors.

The first language-using program that was successful (in public-relations terms at least) was the Eliza system, devised by Joseph Weizenbaum in 1966 and set up to imitate (or parody) a non-directive Rogerian psychotherapist. This meant that much of its discourse consisted of content-free remarks like 'I see' and 'Go on', or of repeating what the person had said in subtly amended form. An example exchange was:

I need some help, that much seems certain.
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP?

Weizenbaum himself was appalled by how seriously many people took what he saw as a clever trick, and consequently became one of AI's sternest critics.

The first useful natural language (NL) system was probably Bill Wood's Lunar (1970). This was a system that enabled geologists to ask questions about samples of moon rocks retrieved by the Apollo missions. Lunar could cope with requests like:

Give me all analyses of sample 10046 for hydrogen.

and

Give me them for oxygen.

where them in the second sentence refers back to all analyses of sample 10046 in the first.

Undoubtedly, the most impressive NL system to date is Winograd's SHDRLU program of 1972. This *tour de force* is perhaps not the breakthrough it seemed at the time, but it produced one of the most memorable man-machine interchanges in print.

25. Why? (Meaning: Why did you pick up the green pyramid?)
    TO GET RID OF IT.
26. Why did you do that?
    TO CLEAN OFF THE RED CUBE.
27. Why did you clean off that cube?
    TO PUT IT ON A LARGE GREEN CUBE.
28. Why did you do that?
    TO STACK UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE.
29. Why did you do that?
    BECAUSE YOU ASKED ME TO.

Winograd's program allowed the user to control the operation of a simulated robot in a micro-world containing children's building blocks by typing at a console in English. Extending his techniques to deal with speech rather than type, and to more complex real-life domains rather than a toy world, has proved difficult. Progress since then has been steady but unspectacular.

The systems described so far have all dealt with the interpretation of textual material. But that is only one of the four principal aspects of language use, each of which has different characteristics.

	Production	Understanding
Text	Writing (very easy)	Reading (difficult)
Speech	Speaking (easy)	Listening (very difficult)

Text is easier to deal with than speech since written language has an existing digital coding scheme whereas spoken language demands difficult acoustic/phonetic transformations. Production of language in either of its forms is considerably easier than its understanding since the latter almost invariably entails filling in a large amount of implicit information. Hence the most challenging of the four tasks is the understanding of spoken language.

It is worth stressing this point because it is possible to obtain devices that recognise words with high reliability (say 96 to 99 per cent) spoken by a small selection of speakers (one to four) from a limited vocabulary (typically 64 to 128 words). There is a great difference between isolated word recognition and continuous speech understanding. Some of the major problems are:

1. Ambiguity: 'rose' is not 'rows' or 'roes'.
2. Background noise: filtering out the surrounding sound.
3. Variations between speakers: regional accents and dialects.
4. Variations in the same speaker over time: happy, depressed, excited tones of voice.
5. Segmentation: 'Onlycomputersputgapsbetweenwords.'

The most serious of these are numbers 1, 4 and 5. Ambiguity is not just a matter of homophones like 'see' and 'sea'. Pronouns such as 'it' and 'him' are inherently ambiguous: their reference can only be disentangled by examining the context of the dialogue. On top of this, there are linguistic ambiguities as exemplified in the phrase: 'Those things over there are my husband's' — just remove the apostrophe and the meaning of the sentence changes entirely.

As for variations in a single speaker, you can see the problem if you imagine that you have installed a voice-lock on your door. During the training phase you might subject it to several examples of yourself by saying: 'Open Sesame'. The mechanism will average them and save the averaged voiceprint as a reference template. All is fine until you rush home one night and breathlessly command: 'Open Sesame!', to which the computer-controlled door will calmly reply: 'Identity violation: access denied'.

The problem of segmentation arises because people run words together. Deciding where to chop up a continuous acoustic signal into words cannot be done on the basis of acoustical information alone. In general, it requires contributions from four sources of information.

A Rows By Any Other Name
The inherent ambiguities of spoken language present massive problems for speech recognition systems: 'rose', 'rows' and 'roes' sound the same, but each has a very different meaning which is only clear within the context of a sentence. For example, the correct version of rose/rows/roes is clear to us from the meaning of the sentence 'the choir boys sat in rows', but it may be difficult to make a computer understand that choir boys do not normally sit in fish eggs

- Acoustic : the speech waveform.
- Syntactic : the rules of grammar.
- Semantic : what makes sense.
- Pragmatic : what the speaker might want.

The Hearsay system, constructed at Carnegie-Mellon University in 1976 and subsequently refined, makes use of all four sources of knowledge. Its task domain is chess, and the user can play the computer by issuing commands through speech.

Each level of linguistic information contributes to the 'understanding' of an input by helping to eliminate implausible hypotheses about what was said. This can be illustrated by an example:

'Queen to King 2'.

On the basis of the sound signal alone, the following utterances are possible.

	Spoken Command	Syntactically Valid?
1.	K to K to	no
2.	K to K2	yes
3.	K to Q to	no
4.	K to Q2	yes
5.	K2 K to	no
6.	K2 K2	no
7.	K2 Q to	no
8.	K2 Q2	no
9.	Q to K to	no
10.	Q to K2	yes
11.	Q to Q to	no
12.	Q to Q2	yes
13.	Q2 K to	no
14.	Q2 K2	no
15.	Q2 Q to	no
16.	Q2 Q2	no

As you can see, it can't reliably tell the difference between 'two' and 'to' or 'K(ing)' and 'Q(ueen)'. However, the system makes the assumption that the utterance was a legal 'chess sentence', and according to the syntax of that restricted language, only items 2, 4, 10 and 12 are grammatical. The rest can be eliminated.

It makes a further (and stronger) assumption that the speaker is trying to play legal chess. Let's suppose, in this particular game, that sentences 4 and 12 are illegal because the player has a piece (a pawn, for example) on Q2 and will, of course, not try to take it. That move fails the semantic test: it is a nonsense sentence in the chess language.

Finally, there are two candidates left (numbers 2 and 10). How does the system choose between them? There may be a preference from the sound signals but it is unlikely to be decisive. In general, it invokes pragmatic information: it makes a *very* strong assumption — namely, that the player is trying to win the game. If it sees that (according to its own chess-evaluating algorithms) K to K2 is virtually suicidal, while Q to K2 is a strong move, it will pick this move as its interpretation.

We conclude this discussion by considering a small example that owes more to Eliza's simple keyword-matching techniques than to more sophisticated later work. The database used in our enquiry program is a table of the 16 largest British companies (1982-83 data). All queries are interpreted by trying to find which firm or which chairman is referred to and which attribute of that company or individual is desired. It can answer questions like:

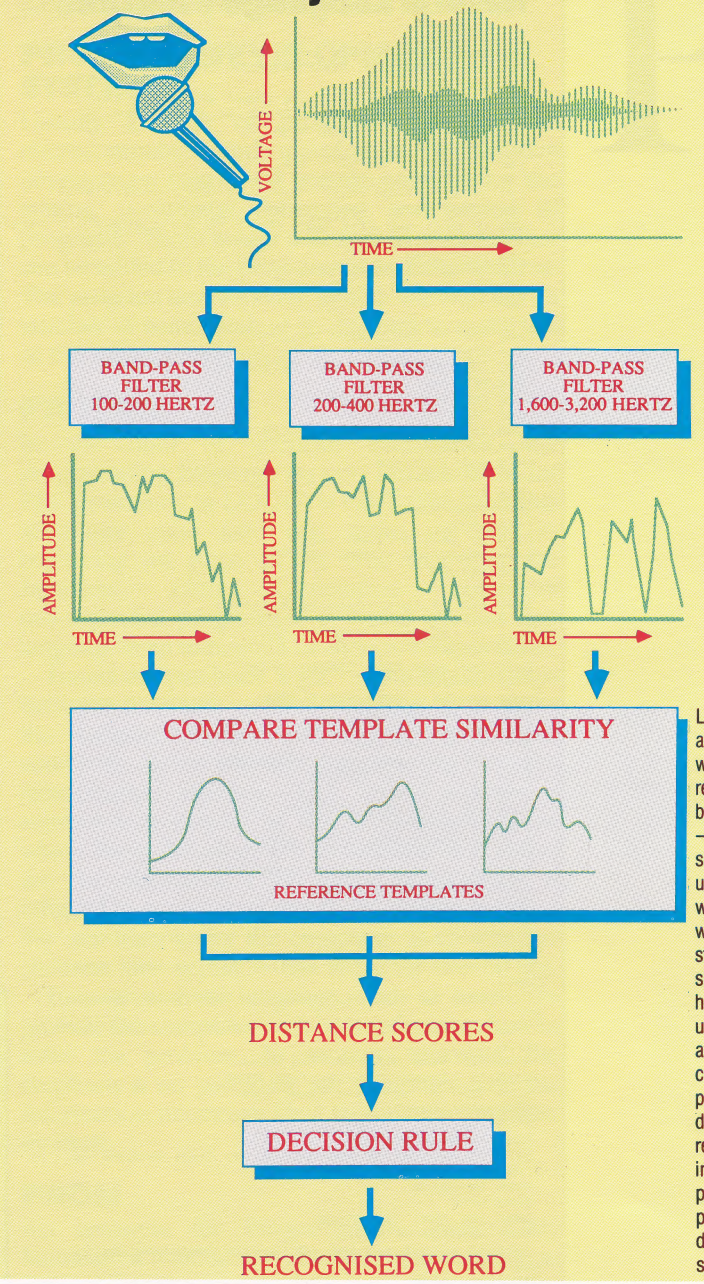
Who is the chairman of ICI?
What was GEC's profit figure?

But it cannot give a sensible reply to more complex questions like:

Did oil companies make bigger profits per unit than tobacco companies?

You may like to consider some of the improvements necessary if it is to be capable of responding to questions of this kind (as the better modern systems do).

What Did You Say?



Language is much more than a collection of individual words, but any speech recognition system must start by recognising words reliably — the most sophisticated semantic analysis is of little use without the raw material, words, to work with. Standard word recognition systems start by dividing the input signal into its low, middle and high frequency components using filters. The resulting acoustic patterns are then compared with reference patterns in memory and distance scores (scores related to how closely the input pattern fits the reference pattern) produced. A decision procedure then analyses the distance scores to classify the spoken input

Natural Language Program

```
  100 GOSUB 1000  : REM read the 'database'
  110 GOSUB 8500:REM READ SYNONYMS DATA
  120 LAST%=0
  125 PRINT"I know a little about UK companies:"
  128 PRINT "Ask me for information about them."
  150 REM **** MAIN LOOP ****
  160 INPUT Q$
  166 IF Q$="" THEN GOTO 220
  170 GOSUB 3000 : REM find subject
  177 IF QT<1 THEN PRINT "Sorry, I don't understand!"
  180 GOSUB 4000 : REM find attribute
  188 IF AT<1 AND QT>0 THEN PRINT "I don't know. I'm afraid."
  190 GOSUB 5000 : REM make answer
  200 LAST%=CO%
  220 IF Q$<>"" THEN 150
  250 PRINT"Bye for now!"
  300 END
  999 :
 1000 REM -- Data Input Routine:
 1010 C%=0
 1020 DIM DB$(9,32) : REM data-base
 1024 RESTORE
 1030 REM **** GET DATA ****
 1040 READ X$
 1050 IF X$="*" THEN GOTO 1110
 1060 C%=C%+1
 1070 DB$(1,C%)=X$
 1075 PRINT X$
 1080 FOR I%=2 TO 9
 1090 READ DB$(I%,C%)
 1100 NEXT
 1110 IF X$<>"*" THEN 1030
 1120 PRINT :C%,"items read in."
 1150 RETURN
 1155 :
 3000 REM -- Topic-finding Routine:
 3010 QT=0
 3020 IF LEN(Q$)<1 THEN RETURN
 3025 CO%=0
 3030 REM **** SEEK COMPANY ****
 3040 CO%=CO%+1
 3050 N$=DB$(1,CO%):GOSUB 3300
 3055 IF SK THEN QT=1
 3060 N$=DB$(2,CO%):GOSUB 3300
 3065 IF SK THEN QT=1
 3070 IF QT<=0 AND CO%<C% THEN 3030
 3080 IF QT>0 THEN RETURN
 3090 CO%=0
 3100 REM **** SEEK CHAIRMAN ****
 3105 CO%=CO%+1
 3110 N$=DB$(4,CO%):GOSUB 3300
 3115 IF SK THEN QT=3
 3120 IF QT<=0 AND CO%<C% THEN 3100
 3132 IF QT>0 THEN RETURN
 3133 IF LAST%=0 THEN RETURN
 3135 N$=" it":GOSUB 3300:IF SK THEN QT=1: CO%=LAST%
 3140 N$=" him":GOSUB 3300:IF SK THEN QT=3: CO%=LA
ST%
 3144 N$=" they ":GOSUB 3300:IF SK THEN QT=1: CO%=
LAST%
 3148 N$=" he ":GOSUB 3300:IF SK THEN QT=3: CO%=LA
ST%
 3150 RETURN
 3190 :
 3300 REM **** SEEK ROUTINE ****
 3325 IF LEN(N$)>LEN(Q$) THEN SK=0
 3330 REM 1st put into lower case:
 3340 A$="":B$=""
 3350 FOR P%=1 TO LEN(Q$)
 3360 X$=MID$(Q$,P%,1)
 3370 IF ASC(X$)>64 AND ASC(X$)<91 THEN X$=CHR$(AS
C(X$)+32)
 3380 B$=B$+X$
 3390 NEXT
 3400 FOR P%=1 TO LEN(N$)
 3410 X$=MID$(N$,P%,1)
 3420 IF ASC(X$)>64 AND ASC(X$)<91 THEN X$=CHR$(32
+ASC(X$))
 3430 A$=A$+X$
 3440 NEXT
 3450 SK=INSTR(B$,A$)
 3455 RETURN
 3460 :
 4000 REM -- Attribute-finding Routine:
 4010 AT=0
 4020 IF QT=0 THEN RETURN : REM no use!
 4030 A%=0
 4040 REM **** FIND ATTRIBUTE ****
 4050 AT=AT+1:WD$=""
 4070 REM **** CHECK EACH SYNONYM ****
 4075 Y%=0
 4080 A%=A%+1:X$=S$(A%)
 4088 IF WD$="" THEN WD$=X$
 4090 IF X$<>"*" THEN N$=X$:GOSUB 3300:Y%=SK
 4100 IF X$<>"*" AND Y%=0 THEN 4070
 4120 IF Y%=0 AND AT<=7 THEN 4040
 4130 IF AT>7 AND Y%=0 THEN AT=0 : REM failed.
 4133 IF AT=0 AND QT=3 THEN AT=1
 4140 RETURN
 4150 :
 5000 REM -- Answering machine:
 5010 IF QT*AT = 0 THEN RETURN
 5020 IF QT=3 OR AT=3 THEN PRINT DB$(4,CO%);" is c
hairman of ";DB$(1,CO%);"."
 5030 IF QT=3 AND AT=1 OR AT=3 THEN RETURN
 5050 PRINT "The ";WD$;" of ";DB$(2,CO%);" is ";DB
$(AT+1,CO%);
 5060 IF AT>3 AND AT<7 THEN PRINT " million pounds
";
 5070 PRINT "."
 5080 RETURN
 5200 :
 8000 :
 8010 REM company information:
 8020 DATA BP, British Petroleum Co.,Oil,PI Walter
s,34583,17306,5589,145150,UK
 8030 DATA Shell UK,Shell Transport & Trading,Oil,
Sir Peter Baxendall,21910,11962,3246,111111,UK
 8040 DATA BAT,"B.A.T. Industries",Tobacco,"P. She
ehy",11318,4607,1018,178000,UK
 8050 DATA ICI,Imperial Chemical Industries,Petroc
hemicals,"J.H. Harvey-Jones",7358,5421,724,123800,
UK
 8060 DATA Shell,Royal Dutch Shell,Oil,JM Raisman,
6665,3704,1206,19027,Holland
 8070 DATA Esso,Esso Petroleum Co.,Oil,AW Forster,
6109,2891,1315,7628,USA
 8080 DATA Unilever,Unilever plc,Food,K Durham,544
7,2434,406,69233,UK
 8090 DATA Imperial,Imperial Group,Tobacco,GC Kent
,4614,1124,192,101300,UK
 8100 DATA P & O,P & O Steam Navigation Co.,Shippi
ng,JM Sterling,4206,927,77,12512,UK
 8110 DATA GEC,General Electric Co.,Electrical Eng
ineering,Arnold Weinstock,4190,2133,621,188802,UK
 8120 DATA Grand Met,Grand Metropolitan,Hotels,SG
Grinstead,3849,2350,366,129454,UK
 8130 DATA RTZ,Rio Tinto Zinc Corporation,Mining,S
ir Anthony Tuke,3680,5163,492,70314,UK
 8140 DATA Ford,Ford Motor Co.,Motor Vehicles,SEG
Toy,3287,2143,278,69500,USA
 8150 DATA British Leyland,British Leyland plc,Mot
or Vehicles,Sir Michael Edwardes,3072,1346,-102,10
5062,UK
 8160 DATA GWH,George Weston Holdings,Food,GH West
on,2981,817,157,72832,UK
 8170 DATA Berisford,S & W Berisford,Commodities,E
S Margulies,2729,788,87,5190,UK
 8440 DATA "*"
 8500 REM**** REM READ SYNONYMS DATA ****
 8510 DIM S$(62)
 8520 FOR P%=1 TO 61:READ S$(P%):NEXT P%
 8530 RETURN
 8540 :
 9000 REM -- synonyms:
 9010 DATA company,firm,organization,corporation,*
 9020 DATA business,activity,trade,trading,industr
y,kind,make,*
 9030 DATA chairman,boss,supremo,director,in charg
e,head,leader,runs,who,*
 9040 DATA turnover,amount,sales,gross,income,how
big,*
 9050 DATA capital,money,share,value,interest,cash

 9060 DATA profit,tax,made,loss,lose,perform,retur
n,gain,how much,*
 9070 DATA workforce,employees,employ,job,people,h
ow many,worker,*
 9080 DATA country,nation,uk,british,foreign,contr
ol,origin,where,*
 9090 :
```

Basic Flavours

The program is written for the BBC Micro. For the Commodore 64 and Spectrum make the following changes:

Commodore 64:

```
3450 SK=0:L=LEN(A$)
3451 FOR T=1 TO L-1
3452 FOR R=1 TO L
3453 IF MID$(A$,T,R)=B$ THEN SK=T:T=L:R=L
3454 NEXT R:NEXT T
```

Spectrum:

Remove all % signs from variable names. Replace LAST% by LA, DB$(,) by D$(,) and WD$ by W$. Make the following changes:

```
1020 DIM D$(9,32,30)
3360 LET X$=Q$(P TO P)
3370 IF CODE(X$)>64 AND CODE(X$)<91
     THEN LET X$=CHR$(CODE(X$)+32)
3410 LET X$=N$(P TO P)
3420 IF CODE(X$)>64 AND CODE(X$)<91
     THEN LET X$=CHR$(CODE(X$)+32)
3450 LET SK=0:LET L=LEN(A$)
3451 FOR T=1 TO L
3452 FOR R=1 TO L
3453 IF A$(T TO T+R)=B$ THEN LET SK=T:LET T=L:LET R=L
3454 NEXT R:NEXT T
4090 IF X$(TO 1)<>"*" THEN N$=X$:GOSUB 3300::LET
     Y=SK
4100 IF X$(TO 1)<>"*" AND YY=0 THEN GO TO 4070
8510 DIM S$(62,20)
```

P

PROGRAM

A *program* is a sequence of instructions given to a computer system intended to make the system perform a task (or series of tasks), which can be executed by the central processing unit. A program should, in theory, arrange these instructions in a logical sequence and in a way that can be understood by the part of the computer that decodes program instructions.

Generally, a program is input via the computer's keyboard and stored somewhere in memory — most computers have a specific area in which programs are stored. What happens then depends very much on the type of program it is and in what format it has been written.

All programs are written in languages. These can range from simple low-level types like Assembly language or FORTH to one of the higher level languages such as PASCAL or COBOL — even the simple set of hexadecimal instructions to the processor can be considered as being a language. All programming languages have common attributes. Each, for example, consists of a precise syntax and vocabulary that cannot be deviated from in the way our spoken languages can.

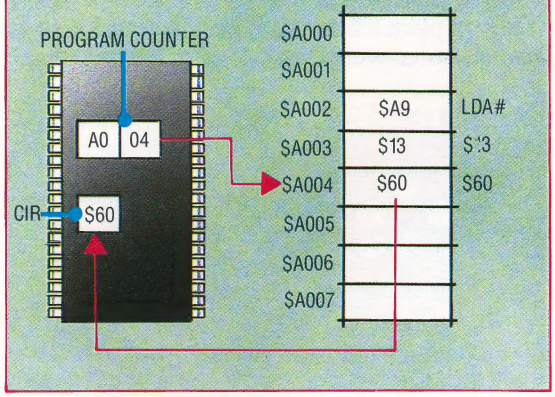
Many programming languages require the source code to be 'compiled' once the program has been written into the computer. This means that the list of instructions is placed in various parts of the computer's memory in order to be executed. Variables will be assigned to their respective positions and labels will be given address locations. Often, any mistakes in the program will be noticed by the compiler and will have to be corrected before it can be run or executed.

PROGRAM COUNTER

This is a register held within the central processing unit that holds the address of the next byte to be fetched from memory. Normally, the *program counter* (also known as the *instruction counter*) is incremented after each byte. However, where the program encounters a branch instruction, the program counter will be modified accordingly.

Now, Where Was I?

The program counter is one of the fundamental components of a computer system. It is particularly important in home micros, which hold parts of a program — such as ROM routines and machine code subroutines – in different areas of memory. So that the computer can 'know' which instruction to execute next, the program counter holds the address of the next instruction to be executed. When this has been accessed, the CPU places it in the CIR (current instruction register)



KEVIN JONES

PROGRAM DEVELOPMENT SYSTEM

Usually a suite of programs intended for microcomputers and mainframes, a *program development system* will contain software for the editing, compiling, debugging and testing of programs. There is also a library of routines that can be added to the program under development. The program development system will also have a number of commands at its disposal to make use of the various facilities of the system.

PROLOG

In the field of artificial intelligence, the programming language PROLOG (an acronym for PROgramming in LOGic) is one of the principal languages used (see our series on PROLOG beginning on page 1272). The fact that the Japanese Fifth Generation computer development programme is based around this language is an indication of PROLOG's importance.

Based on 'predicate logic' — the relationships between constants — PROLOG allows us to make the computer deduce information about these constants. As an example, suppose we have the relationship 'children like apples'. This is expressed in PROLOG as like (children, apples), in which like is the predicate and children and apples are the constants. (The entire relationship is referred to as an 'atom' in PROLOG nomenclature.) We can then query the computer as to whether children like apples — the response will be yes.

In a procedural language, such as BASIC, we provide the computer with instructions on *how* to solve the problem. In PROLOG, however, the emphasis is on providing the computer with the information necessary for it to solve the problem by itself.

PROM

An acronym for Programmable Read Only Memory, a *PROM* is a semiconductor device in which the contents of the ROM are added after the device has been manufactured. The data or programs are added at a later date by passing currents from a machine, known as a *PROM programmer*, through the matrix that makes up the device. A voltage will be passed down the address pins of the PROM and then a much larger voltage will be sent down the data lines and seal the bit pattern in place at that address by fusing the links. Thus the data held in the addresses that make up the memory will be fixed.

Although some PROMs, such as 'EPROMs' (Erasable PROMs, see page 1044) can be erased by bombarding the chip with ultraviolet light (allowing them to be used again), the contents of most are not alterable. PROMs are widely used to test devices that are used in the manufacture of computers, and also by hobbyists who wish to store their own programs in ROM.

READY, STEADY...

Besides chess, the ancient Oriental game of Go has intrigued AI researchers because of its subtly complex qualities. We begin a project to develop a program to play Go, by first outlining the rules of the game and introducing some of its infinite nuances.

Good chess-playing programs are now quite common and researchers in AI have turned to more complex games. One of these is the Oriental game of Go. Though the rules of the game are far simpler than chess, it is often regarded as being far more complex. After more than 20 years of research, the top programs still play at little better than novice level.

During the series we will be developing a program to play the game. Though not highly proficient, the program provides a good introduction to the game, and should prove to be a worthy opponent for newcomers. The program has been specifically designed so that it may be easily modified and developed.

Go is a two-player game, played on a board with 361 intersections, as shown in figure 1. The two players, one playing with black stones and the other with white, alternately place one of their stones on any vacant intersection of the board. Note that the stones are placed on the junctions of lines and not in the squares.

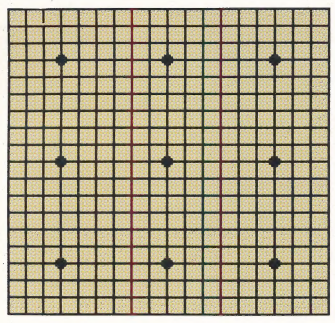
HOW TO PLAY GO

The object of the game is to surround territory (vacant intersections) with your own stones. To win, your stones must be surrounding the most territory at the end of the game.

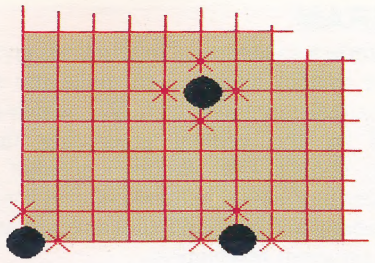
The first move in Go is always made by Black, who is usually the weaker player. If there is a significant difference between the playing strengths of the two players, Black's first move will consist of placing between two and nine 'handicap' stones onto the board, positioning them in a specific pattern onto the nine points marked on the board, as shown in figure 1.

Merely surrounding territory is a bit too easy — stones can also be captured and removed from the board. Any vacant point immediately adjacent to a stone (along a line) is called a *liberty*. Thus any single stone can have two, three or four liberties depending upon its position on the board (see figure 2). In order to capture an opposing stone, you must place your own piece on all the opposing stone's liberties. Thus in figure 3, if Black's last move was to fill the last liberty of the white stone, the latter would be removed from the board. All captured stones are added to the player's score.

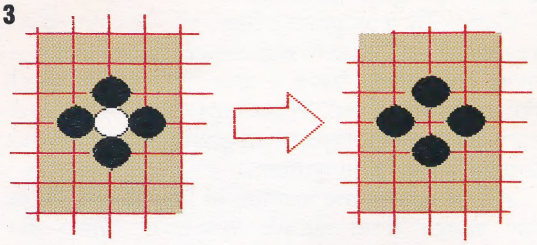
1



2

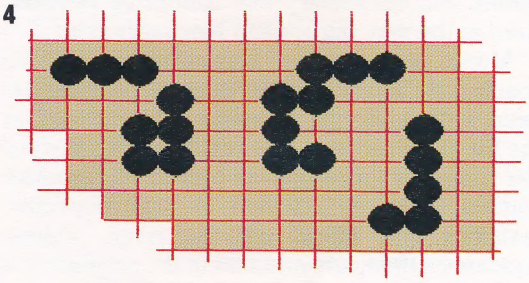


3



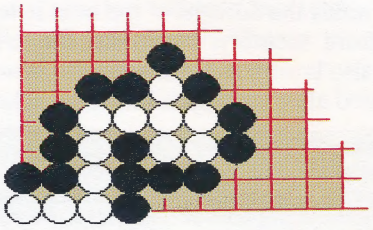
If one or more of a stone's adjacent points are filled with other stones of the same colour, they are said to be connected and form a *group*. Diagonal connections do not form links between stones: the following figure, for example, shows four groups, not three:

4



Groups are more difficult to capture than single stones because they must be removed as single units. Thus Black might have to fill 15 liberty points in order to capture a group of white stones.

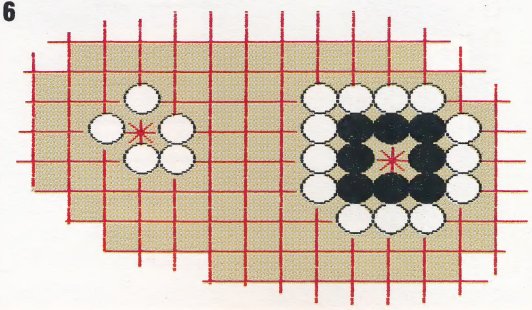
5



This is really all you need to know in order to play the game, but there are a few complications that derive from the rules we have given:

Suicide: As the rules stand, it is possible for a player to place a stone such that it has no liberties. For instance, with Black to play, either of the

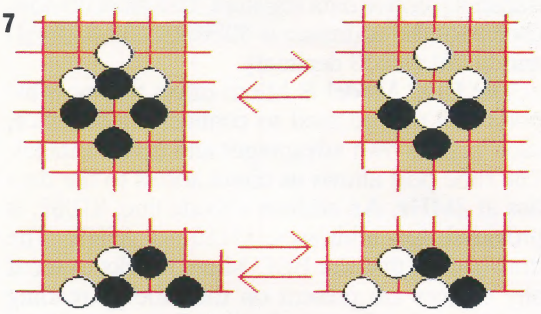
6



points marked in figure 6 would cause this situation. This move, known as 'suicide', is prohibited in Go.

Ko: This means 'infinity' and refers to a possible situation in which stones could be placed and captured indefinitely. The two diagrams following are common occurrences of this.

7



To cater for this situation, the rule of Ko prohibits a player from placing a stone such that 'the immediately prior' position is produced. This rule brings into effect what are known as 'Ko fights'. If White is prevented from immediately recapturing a stone because of the Ko rule, then obviously an intermediate stone must be played.

This stone will usually be placed in a position that threatens either a black stone, or group, elsewhere on the board, and is known as a *Sente* (forcing a reply). Thus Black cannot fill the Ko point, which ends the Ko fight, and White is able to recapture the black Ko stone. Black is now prevented from immediately recapturing and must try to find a Sente move if the Ko fight is to continue.

'LIFE AND DEATH'

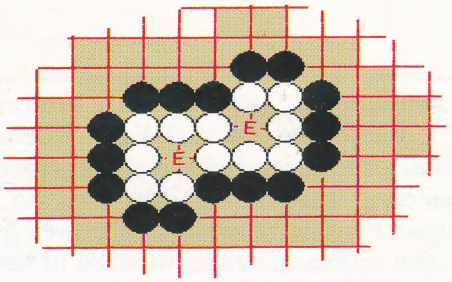
A very important concept resulting from these rules is that of 'Life and Death' for a group of stones. We have already seen that it is impossible for Black to play on the vacant intersection in the middle of the group (see figure 6). Notice, however, that if White were to play here, the black group would be captured.

It would also eventuate that if the black group was not surrounded by white stones, then White could not place a stone on this point. It follows that if White wants to capture this black group, the very last white stone placed must be on this vacant point after having entirely surrounded the black group with white stones. The black group would then be removed from the board, giving the white stone just placed four liberty points.

A point that is surrounded by stones of the same colour in this way is known as an *eye,* and it is always the case that the eye must be the last liberty filled to capture the group.

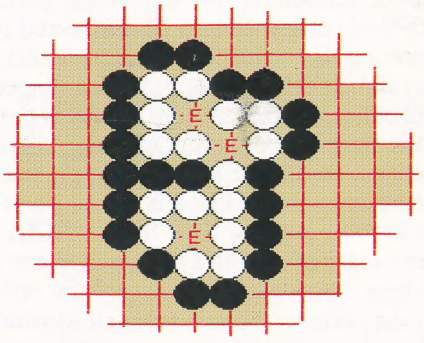
By taking this concept one stage further, we can imagine a group containing two eyes, such as the white group shown in figure 8. To capture the group, the last black stone must fill both eyes simultaneously — but this is impossible, because Black can only place one stone at a time! Consequently, this group (and any containing at least two eyes) is safe from capture, and will remain on the board until the end of the game (providing, of course, that White is not stupid enough to fill the eyes!).

8

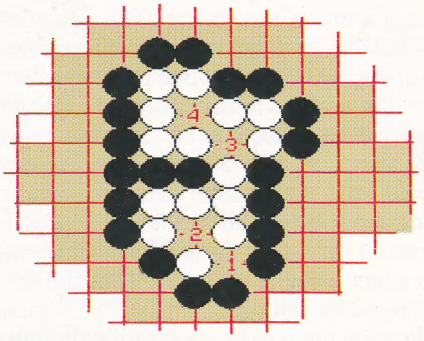


It is not even necessary that the eyes be contained in one group. The three white groups in figure 9 share between them three eyes, and all white stones are safe from capture. You must, however, be careful. The similar formation in figure 10 (only one stone different) can be captured by Black using the sequence shown. The lowest of the three eyes can, in fact, be infiltrated by Black — this is known as a *false eye.*

9



10



All of these groups have had fully-developed eye formations, but in practice it is not necessary to form two eyes for each group, but merely to have the potential to do so if attacked.

The game ends by mutual consent of the players, when both feel there is no more to be gained on either side. This rather ambiguous ending has caused many problems for Go programs, which still find it very difficult to decide when the game should finish. The game is also deemed finished if both players 'pass' their move, returning control to the opponent. Passing is allowed by either player at any stage during the game, but is rarely used.

Who Wins?

At the end of the game, each player's score is calculated in the following manner:

1. All neutral points, known as 'dame', are filled. Either player can fill these, since they will not be calculated in the final score.

2. Any stones or groups that cannot avoid capture are removed from the board, as if they had been captured. This implies that it is not necessary to capture these 'dead' stones during the game, as long as you are able to do so if you wanted. The capturing sequence could be played out, but would benefit neither player. (Each defending stone placed would increase the attacking player's score by one point, but the attacking player would also have to place a stone, thus filling his territory, and reducing his score by one point.)

3. Each player's score is then calculated as the number of vacant intersections (the territory) controlled by the player, minus the number of stones captured by the opponent. The winner is the player with the most points.

MELODIC LINES

We continue our MIDI interface project for the Commodore 64 and BBC Micro by outlining the second phase of construction: connecting the ACIA chip pins to the computer's data, control and power lines. We also outline a simple method of testing the finished board from BASIC.

The Asynchronous Communications Interface Adaptor (ACIA) chip forms the basis of the MIDI interface and is compatible with the 6502 and 6510 processors used by the BBC Micro and Commodore 64, respectively. We can therefore link the processor's data, address and control lines directly to the chip pins.

The finished board is designed to plug directly into the Commodore 64's expansion port. The BBC Micro version should be connected to the tube port on the underside of the computer via a 40-way ribbon cable and appropriate connectors. We will outline the connections that must be made for each micro separately. Once the board has been constructed, there are some simple tests that can be made to ensure that it is working correctly. These involve placing data in the ACIA chip registers and performing a simple loop test.

Address decoding is necessary in order to access any peripheral that is connected to the main data bus, but which does not possess the same number of address lines as the CPU. Such devices usually have a 'chip select' pin that enables the device to use the data bus when the chip select line is active. The address lines connected to the device will then allow selection of different registers.

Chip select signals for the various devices connected to the system are generated by decoding the most significant address lines. Each combination of bits on these address lines can be used to select a device uniquely. Thus the device's internal registers will appear in the CPU's memory map, allowing them to be accessed as though they were normal memory locations. However, care must be taken to ensure the addresses used by the device do not correspond to important registers used by the host computer's operating system.

The Commodore 64's expansion port has two outputs, labelled I/O1 and I/O2. These lines go low when pages $DE and $DF, respectively, are accessed. By simply connecting the CS2 line on the ACIA chip to I/O1, we can map the chip into page $DE. Because the ACIA is not connected to the address lines A1 to A7, the chip's internal registers can be accessed by any address in the range $DE00 to $DEFF. By connecting A0 directly to the ACIA's register select pin, all even addresses will access the transmit/receive data registers. The most obvious choice for the addresses is $DE00 (56832 decimal) and $DE01 (56833 decimal).

The BBC Model B Micro offers two possible ports that can be used to connect the interface, each with its own advantages and disadvantages. The tube port allows us direct access to the data bus at 2MHz. An address decode line, NTUBE, is provided for addresses &FEE0 to &FEFF. The drawback is that the BBC Micro checks to see if any devices are present on the tube by reading certain tube addresses when it is first powered up. If anything other than the second processor is attached, the computer will appear to hang up as it waits for data from the second processor.

A professional answer to this problem would be to connect one of the high-order address lines to CS0 on the chip. An alternative is to connect NTUBE to CS2 and plug the board into the tube port with the computer on. For simplicity, this is the solution chosen for this project. However, a further alternative is to use the 1MHz bus. Pins marked NPGFC and NPGFD are provided to map external devices into pages &FC and &FD and could be used in the same way as I/O1 and I/O2 are used in the Commodore 64 version.

There are two disadvantages with using the 1MHz bus: first, because the system clock is slowed down to 1MHz, 'glitches' are present on the two decode lines. A clean-up circuit (such as that described in section 28.5 of the advanced user guide for the BBC Micro) must be used to avoid this problem. The second problem is that there is no 5v power feed to the 1MHz bus. Power must therefore be derived from either the user port or the auxiliary power connector.

Bit Wise

If you wish to program the MIDI interface yourself it is important to understand the functions of the four ACIA chip registers. We continue the discussion of these functions — which we began on page 1345 — and illustrate the bit functions of the control and status registers diagramatically.

The ACIA can be programmed via the control register to interrupt the CPU when certain status bits become set in the receive and/or transmit sections of the ACIA.

A transmitter interrupt will occur if control register bits 5 and 6 are set and clear, respectively, and the TDR status bit is set. The interrupt is cleared by writing data to the TDR.

A receiver interrupt occurs if both control bit 7 and status bit 0 are set. Reading the RDR will clear the interrupt (unless status bit 5 is also set, which is

The ACIA Registers

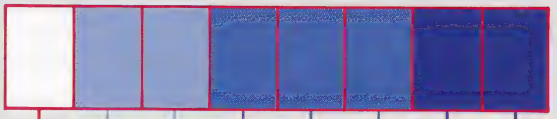
TRANSMIT/RECEIVE DATA REGISTERS:
REGISTER SELECT = 1

RECEIVE — READ REGISTER
TRANSMIT — WRITE TO REGISTER

STATUS REGISTER (READ ONLY) REGISTER SELECT = 0

- RECEIVE DATA REGISTER FULL
- TRANSMIT DATA REGISTER EMPTY
- DATA CARRIER DETECT
- CLEAR TO SEND
- FRAMING ERROR
- RECEIVER OVERRUN
- PARITY ERROR(*)
- INTERRUPT REQUEST

CONTROL REGISTER (WRITE ONLY) REGISTER SELECT=0



COUNTER DIVIDE SELECT
00 DIVIDE BY 1 (*)
01 DIVIDE BY 16 (*)
10 DIVIDE BY 64
11 MASTER RESET

WORD SELECT

	DATA BITS	STOP BITS	PARITY
000	7	2	EVEN (*)
001	7	2	ODD (*)
010	7	1	EVEN (*)
011	7	1	ODD (*)
100	8	2	NONE (*)
101	8	1	NONE
110	8	1	EVEN (*)
111	8	1	ODD (*)

TRANSMIT INTERRUPT CONTROL
00 DISABLED (*)
01 ENABLED
10 DISABLED (*)
11 DISABLED (*)

RECEIVE INTERRUPT CONTROL
0 DISABLED
1 ENABLED

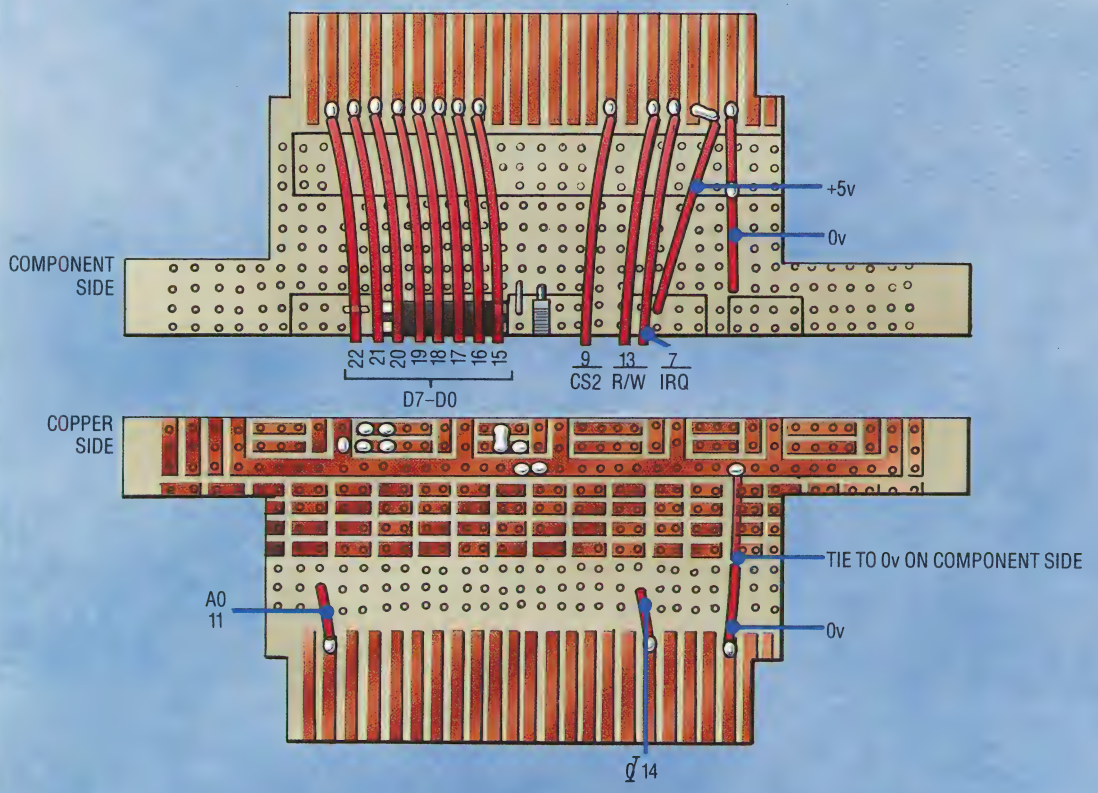
(*) = NOT APPLICABLE TO MIDI PROJECT

KEVIN JONES

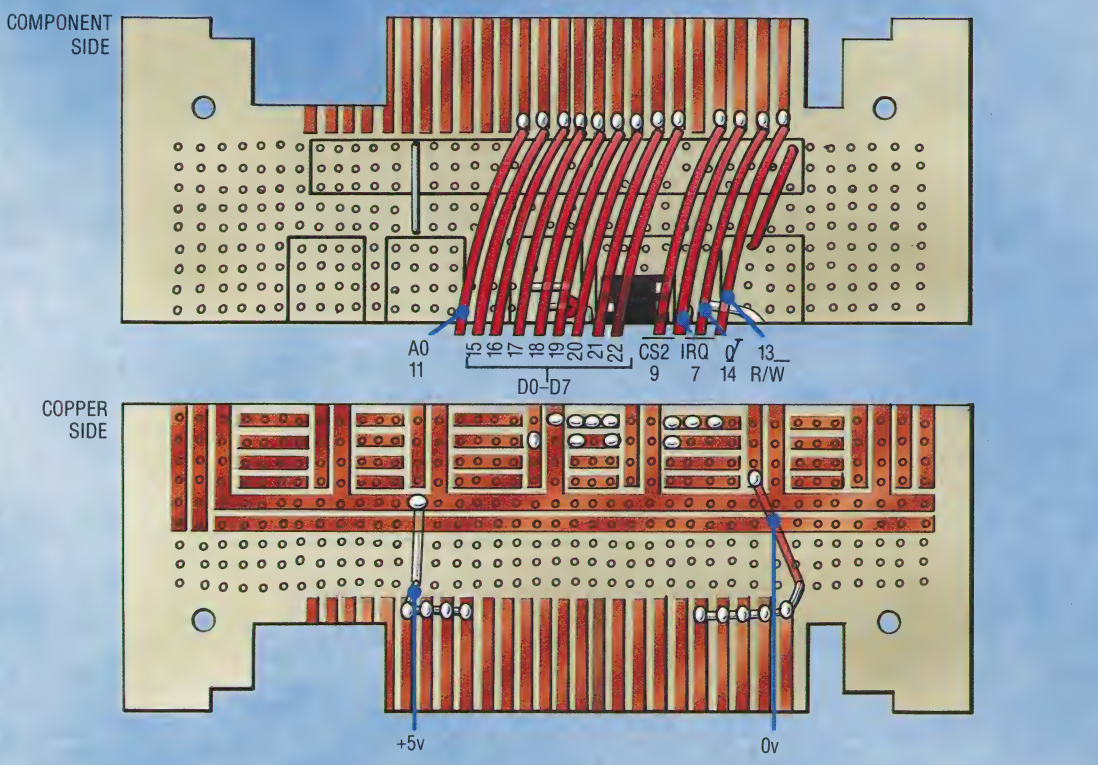
cleared by reading the status register before the RDR). Status bit 2 can also generate an interrupt, but it is not applicable in this project. Separate lines are provided for receive and transmit clocks, but they will usually be connected together. The baud rate clock is derived directly from the clock input or by dividing by 16 or 64, depending on the state of control bits 0 and 1.

Bits 2, 3 and 4 in the control register determine the number of stop bits, the number of data bits and whether parity is even, odd or not in use. For MIDI, we require bits 2 and 4 set and bit 3 clear (eight data bits, no parity, one stop bit) to conform to the standard. Bits 5 and 6 in the control register handle transmitter control. For MIDI, we require bit 6 to be clear and bit 5 will then enable transmitter interrupts

Commodore 64



BBC Micro



KEVIN JONES

Edge Connections

The diagrams here show the appropriate connections to the ACIA chip pins for each micro. The leads from the edge connectors are numbered. These relate to the pin numbers on the ACIA chip. Note that this chip's pins are numbered as follows: with the chip held notch uppermost and the pins away from you, pin 1 is nearest the notch on the left-hand side. The pins are then numbered down the left-hand side to 12. Pin 13 is opposite pin 12 on the right-hand side, and the remaining pins are numbered by working up the right-hand side to pin 24, nearest the notch. Use wire-wrap wire to make the appropriate connections.

Once finished, the Commodore 64 version should be inserted, component side uppermost, into the 64's expansion port, and the computer switched on ready for testing. For the BBC Micro version, it is necessary to make up a connection cable using the one metre length of 40-way ribbon cable, a 40-way IDC pin connector and a 20-way edge connector

Testing The Board

With the board plugged in, we can now perform some simple tests to verify correct functioning of the board. A simple multimeter will be useful to isolate any faults in the unlikely event of the board failing any of the tests. If you haven't got one, you will need to perform a close visual inspection of the board.

1. If the computer refuses to function normally with the board inserted, check the following:

- Verify the voltage between the +5v and 0v lines on the board is indeed 5v. If not, check that all the ICs are inserted correctly. Examine the board for short circuits between the power rails.
- Remove the board from the computer and use the multimeter to check for short circuits between any ot the computer bus connections.

2. When the computer appears to be functioning normally, connect a MIDI cable between the MIDI IN and MIDI OUT sockets using standard 5-pin DIN hi-fi connecting cables. Type in the following command (BBC Micro equivalent in brackets):

```
POKE 56832,3    (?&FEE0=3)
```

This command places a 3 in the ACIA control register, which performs a master reset. Now type:

```
POKE 56832,22    (?&FEE0=&16)
```

This places the value $16 in the control register and configures the ACIA as follows:

- Disables receiver and transmitter interrupts (because we have no means of handling them yet).
- Defines the transmitted and received serial words as eight data bits plus one stop bit with no parity generation/checking.
- Defines the baud rate as the clock at pins 3 and 4 divided by 64. (2MHz/64 = 31.25KHz, which is the specified rate for MIDI).

The ACIA should now be ready to receive and transmit data. Check this by reading the status register with:

```
PRINT PEEK(56832)
(PRINT?&FEE0)
```

You should read the value 2. This indicates that both transmit data register (bit 1 set) and the receive data register (bit 0 clear) are empty. Since no data has been received and interrupts are disabled, the remaining status bits 2 to 7 should be clear.

3. Send a byte from the transmit register through the cable and into the receive register with the command:

```
POKE 56833,X    (?&FEE1=X)
```

where X is any number from zero to 255. This places a value in the data register, to be transmitted.

4. In a fraction of the time it takes to type the next command, the byte should have been received. Read the status register again:

```
PRINT PEEK(56832)
(PRINT?&FEE0)
```

The value should now be 3. Bit 1 will have cleared immediately after the last command (transmit register full) but will be set a short time afterwards, as soon as the transmit data register becomes ready for the next byte. Bit 0 will be set indicating that the byte has been received and can be read from the data register. Note there is a possible error: if bit 0 is not set, there is probably an open circuit in the transmission path that is holding the receiver input at a high level preventing from being detected.

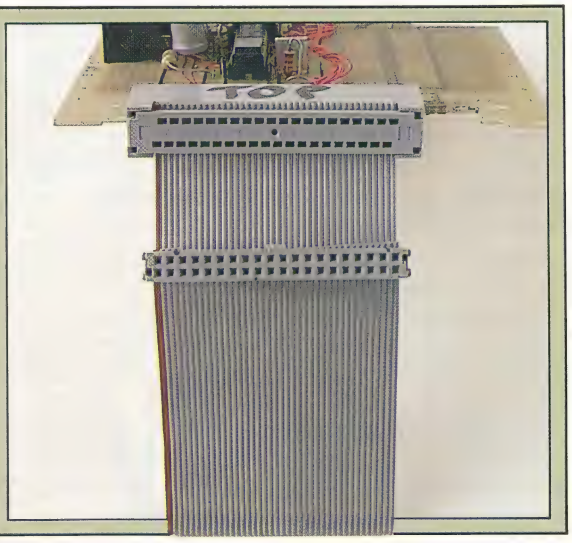
5. Having verified that a byte has been received, we can check to see if it is the same as the one transmitted by reading the data register:

```
PRINT PEEK (56833)
(PRINT ? &FEE1)
```

This read operation should return the same value X as was transmitted before. Steps 3 to 5 should be repeated with various values of X

Ribbon Development

The MIDI interface board can be linked to the BBC Micro's tube port via a 40-way ribbon cable, a 40-way IDC edge connector that fits onto the edge of the board and a 40-way IDC pin socket. This cable plugs directly into the tube port on the underside of the BBC Micro. The cable and connectors are easy to assemble but it is important to note the orientation of the two plugs. Lay the cable flat and attach the edge connector so that pin 1 (marked on the connector housing) is on the bottom row. Mark the upper surface of the housing in some way to indicate the orientation of the connector when it is used with the interface board. The IDC pin socket should be attached at the other end of the cable with the rectangular notch uppermost, as shown



FAST MOVER

Commodore 64 owners have had to use, for the most part, Commodore's own storage devices, which lack both performance and storage capacity. With the introduction of the Phonemark 8500 Quick Data Drive, which uses stringy floppy wafers, the situation has dramatically improved.

CHRIS STEVENS

Front Runner
There have been few 'third party' suppliers of alternative storage systems to the Commodore cassette deck and disk drive, despite the criticisms of these machines. The Quick Data Drive is a 'stringy floppy disk' system that uses continuous loops of tape identical to those in the Rotronics Wafadrive. At around £100, the system is not particularly cheap, but it is half the price of the Commodore disk drive, and in many applications is considerably faster

Because many home computer enthusiasts are considering purchasing faster and more efficient methods of accessing their programs, manufacturers are producing a wide range of mass storage devices aimed at the most popular home micros. Among these is the Phonemark 8500 Quick Data Drive from Dean Electronics, made for the Commodore 64 and Vic-20.

This device is a close relative of the Rotronics Wafadrive (see page 1129) designed for the Sinclair Spectrum. The drives for both were produced by BSR Electronics and each uses identical 'wafers'.

The standard Commodore 1541 disk drive, like the cassette drive, is notoriously slow. This is due not to the disk drive itself but the method by which the computer loads data. Much of the Commodore 64's operating system is inherited from the PET business machines of the mid-1970s, when mass storage systems, particularly cassette drives, were extremely unreliable as data storage devices. When Commodore turned its attention to providing back-up storage for the PET series, it decided to provide its own cassette drive and incorporate a complex series of checks to ensure that the data being loaded was correct. While this increased the reliability of LOADing, it was at the expense of access speed. This system was carried over to the Vic-20 and later the Commodore 64.

Nowadays, the quality of cassette tape has greatly improved and the need for the long and complicated data checking on the Commodore machines has become unnecessary. Many commercial software packages now contain their own loading techniques, which remove many of the checks and make loading much faster, without any loss of reliability.

However, when LOADing their own programs, most users do not have access to these high-speed techniques and must still put up with the delays imposed by the Commodore operating system. The Quick Data Drive, which claims to load 15 times faster than normal cassettes and faster than the 1541 disk drive, can be seen as a commercial alternative to the Commodore peripherals.

The Quick Data Drive is quite a small device — just over half the size of the Commodore 1530 cassette deck. The drive is connected to the Vic-20 or Commodore 64 via the cassette edge connector. Unlike the Rotronics Wafadrive, the Quick Data Drive has only a single drive. While it would be preferable to have two drives when a system wafer and a data wafer need to be running simultaneously, most users will find the single-drive adequate. If absolutely necessary, a second drive can always be purchased and connected to provide dual-drive capability.

THE OPERATING SYSTEM

The fact that the drive plugs into the cassette port does not mean that you cannot have a cassette deck running at the same time. There is a cassette edge connector on the back of the Data Drive allowing cassette drives, or a second Quick Data Drive, to be daisy-chained together.

Although the Rotronics Wafadrive and the Quick Data Drive have similar appearances, they have very different methods of operation, reflecting the differences in the computers for which they were designed. Whereas the Wafadrive has its operating system held in ROM (emulating the ROMs on the Interface 1), the Quick Data Drive Operating System (QOS) is held on a wafer. In order to LOAD the QOS into the computer, you must press Shift/Run (as you would when LOADing a normal cassette). When the screen

prompt PRESS PLAY ON TAPE appears, you then press a small button on the back of the drive that autoboots the system. Once this has been done, the QOS will perform it automatically for subsequent accesses.

The programs that make up the QOS are loaded into two separate areas of memory. First, the machine code routines for LOADing, SAVEing and searching programs are stored between addresses C000 and CFFF towards the top of memory (normally used for machine code programs). It is this module of the QOS that results in the increased LOADing speed of the drive.

Although the QOS does not implement any commands of its own (using instead those already available in the Commodore Operating System ROM), it will intercept the routines that deal with normal loading and insert its own.

QUICK DATA DRIVE

PRICE
£99.99 (inc VAT)

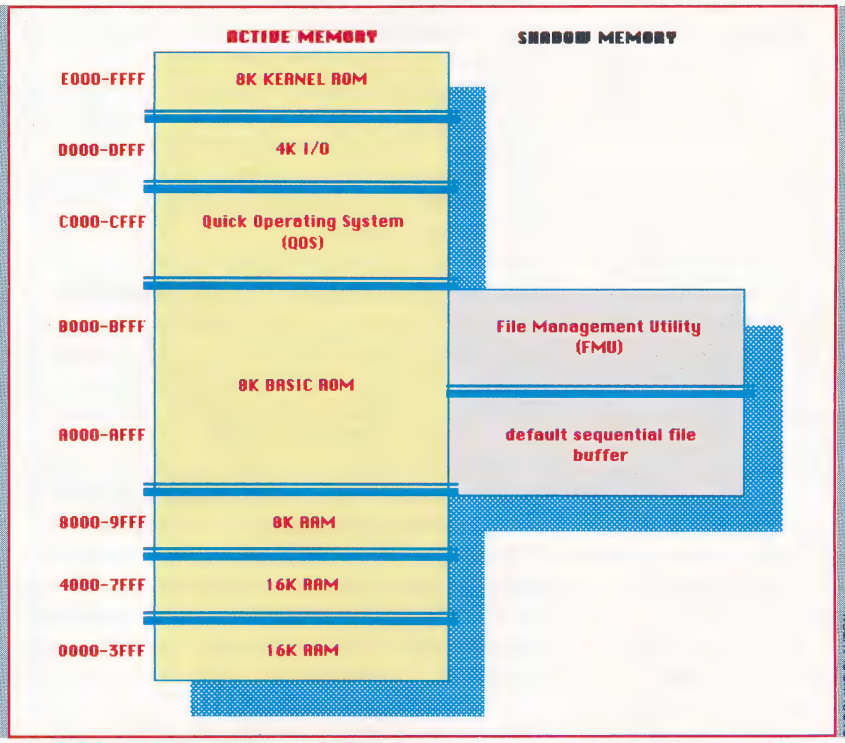
DIMENSIONS
147×118×49mm

INTERFACES
Socket to connect to the Commodore 64/Vic-20 cassette port

FORMAT
Continuous loop, stringy floppy wafers

CAPACITY
16, 64 and 128 Kbyte wafers available

SPEED
Average access time: 8 secs/15 Kbyte file; 43 secs/120 Kbyte file



CAROLINE CLAYTON

The other part of the Quick Operating System is the File Management Utility (FMU), which contains a number of useful routines. This is kept in the top half of the eight Kbytes of Shadow Memory underneath the BASIC ROM — between addresses B000 and AFFF. The bottom four Kbytes between A000 and AFFF are used as a sequential file buffer that is used by the FMU.

Because the FMU is stored in the area of memory under the BASIC ROM, you cannot use both at the same time. In order to call it from memory, the command LOAD 'FMU' must be executed. This simply 'banks' out the BASIC ROM and enters the FMU.

Undoubtedly, the Quick Data Drive's operations are much faster than standard cassette methods. As an example, the New World simulation game featured in our Programming Projects series (beginning on page 1026) was used as a benchtest. The complete 25 Kbyte program, requiring over nine minutes to load from cassette and one and a half minutes from disk, took a mere 30 seconds to LOAD from the Quick Data Drive — a considerable improvement. However, as with all tape loop-based systems, much depends upon where the drive's read/write head is in relation to the beginning of the program.

The Quick Operating System finds the required data files by checking each of the heading blocks on the tape. When a wafer is formatted, the operating system will divide the tape into blocks, each with its own filename section. When it is required to load a file into the computer, it will search until it finds the block containing the first part of the file, load it and then find the second.

Similarly, when asked to display a directory of the wafer, the QOS will read each of the filenames as the read/write head passes over the tape, and register the blocks that contain files and those which are empty. When all the filenames have been read, the system will display the list of files plus the total amount of available space in bytes.

FILE MANAGEMENT UTILITY

The File Management Utility is a menu-driven set of routines covering such applications as formatting and reading the directory of the wafer. It also contains copying routines that allow data to be transferred from disk, cassette or wafer onto a back-up wafer. This is obviously an important feature of the Quick Data Drive system as few programmers would consider buying a storage system, no matter how good, that could not transfer existing programs onto the new medium.

Of course, there are drawbacks to the system. While the copying routines work fine for BASIC programs and sequential files, certain routines (machine code in particular) which are loaded to a specific area of memory will cause problems. This is because the two areas used by the QOS and FMU are also the areas in which machine code is kept. Thus when machine code programs are loaded, they will overwrite the Quick Data Drive operating system and crash the program.

This design feature means that while it is possible to copy your own programs, it is not, as yet, possible to transfer commercial software. If the Quick Data Drive sells in sufficient quantities, however, there will doubtless be a program produced to do so.

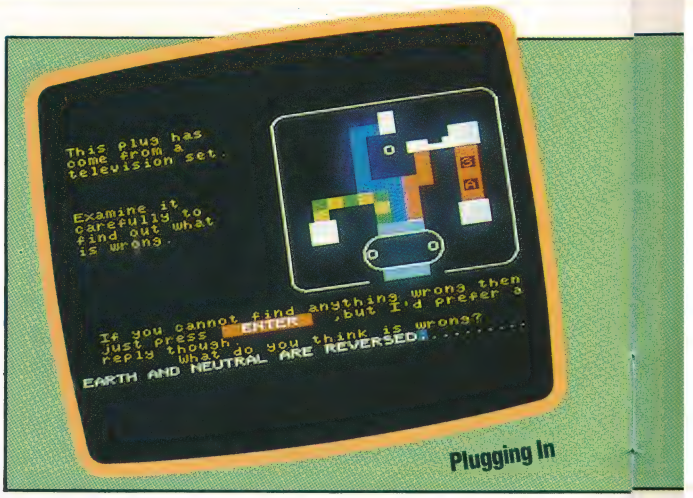
Until then, much depends on Dean Electronics' ability to persuade software houses to sell their games and applications on wafers. In this respect, they may find that the Rotronics Wafadrive will prove a valuable ally — software companies will thus be able to purchase larger quantities of wafers at larger discounts when they find that they can place both their Commodore 64 and Sinclair Spectrum software on the same storage medium.

Although it is still early, the company has had a measure of success. Some games, such as Epyx's highly rated Impossible Mission (see page 1220), are already available on wafer, but a much wider software base will be necessary.

A Quick Memory

The Quick Data Drive does not keep its operating system in ROM but has to LOAD it from a systems wafer. The various components are installed in two separate areas of memory. The first, the Quick Operating System (QOS) is LOADed into the area usually reserved for machine code programs. The other part of the system, the File Management Utility (FMU) is held in the 'shadow RAM' behind the BASIC ROM. Also contained here is a file buffer where programs are loaded before they are SAVED onto another medium

CODE CREATING CODE


This plug has come from a television set.
Examine it carefully to find out what is wrong.
If you cannot find anything wrong then just press ENTER, but I'd prefer a reply though. What do you think is wrong?
EARTH AND NEUTRAL ARE REVERSED.
Plugging In

In this first part of a two-part series on program generators, we look at the benefits of using such a system. We present an overview of four packages, and take a detailed look at the operation of two of them — Sycero and The Last One.

A program that takes the hard work and frustration out of programming, finishing up with lines of code that execute perfectly, first time, without debugging or syntax error messages, would seem like a dream to every first-time programmer (and probably to more experienced ones as well).

In a sense, of course, everyone who writes a BASIC program is already using a program-writing program: whether it is resident in ROM or LOADed in from disk or tape, the BASIC interpreter takes lines typed in by the user and re-writes them into machine code that the computer can understand. This is a much easier process than entering the seemingly meaningless digits of machine code itself. Even if you have progressed to Assembly language programming, a similar process occurs when the Assembly mnemonics are translated into machine code.

Most programmers will be familiar with the trick of making a program generate extra program lines either by using the keyboard buffer with an incrementing counter for the line-numbers, or by POKEing them into memory. This is often used by machine code programmers to PEEK the memory locations and write them into a loader program as DATA statements.

Some languages, for instance COMAL (see page 1122), have built-in error-checking and will refuse to accept a line of code that is incorrectly entered, so that typing PRNT when you meant PRINT will not get through. FORTH will refuse an instruction that uses a word not in its basic vocabulary or not previously defined by the user. LOGO has a similar capacity to stay within its own limitations.

However, the facilities just noted will still not enable you, for example, to instruct the computer to write a program to calculate your income tax. But the growth of systems that mimic or appear to incorporate some level of artificial intelligence makes this seem less remote a possibility. There is a program, for instance, called Microtext, which can be used to guide a non-technical user through complex technical tasks. It can also be used to generate interactive teaching aids, fault-finding programs, questionnaires and information retrieval software. By using an 'authoring' system to create an application, with an active screen editor available during testing, the resulting software will then be 'published' and turned into a run-time only system that cannot be modified by the user.

Microtext is generally available only for the more expensive CP/M machines — making the availability of a sophisticated screen editor a valuable facility — but it has also been made available at a much lower price (under £350) for the Tatung Einstein.

THE QUILL AND ILLUSTRATOR

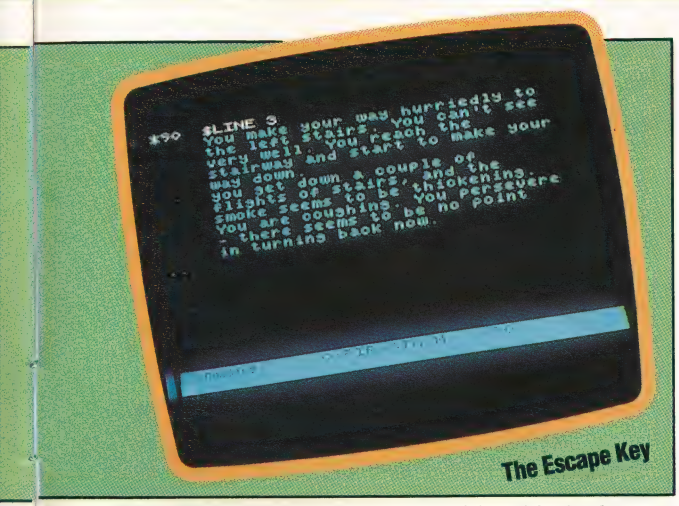
Another remarkably sophisticated program-generator is The Quill (see page 42), specifically designed for those writing adventure games. This allows you to set up and edit a database of information — locations, objects, how they can be manipulated and so on — and incorporate it into an Assembly language program that should RUN first time without crashing.

The Quill has recently been joined by Illustrator, which adds graphics to text adventures. The Quill is available for the Sinclair Spectrum, Amstrad 464, Commodore 64 and Oric, and Illustrator is available for the Spectrum and Amstrad, and will soon be produced for the Commodore machine as well. Despite its low cost, The Quill is good enough to have been used to write many commercial adventure games, most notably Hampstead, by Melbourne House.

Though both are technically program-generators, to the micro industry there are just two programs that actually write programs: the rather boastfully-titled The Last One, which, when it was announced, was touted as the last software you would ever need to buy, and the rather more powerful (though slightly less friendly) Sycero.

TLO, as its distributors now prefer to describe it, was originally written for 64 Kbyte CP/M machines. This is evident as it has the same sort of constant reference to disk overlays that makes WordStar so slow. It is now available for MS-DOS machines with a minimum of 256Kbytes of RAM, but while this permits many more sophisticated features to be added, it still runs at a slightly sluggish pace.

Its writers claim to have incorporated some level of artificial intelligence in the program.


*50 $LINE 3
You make your way hurriedly to
the left stairs, you can't see
very well, you reach the
stairway and start to make your
way down.
you get down a couple of
flights of stairs, and the
smoke seems to be thickening.
You are coughing, you persevere
there seems to be no point
in turning back now.
The Escape Key

Generation Games
These screens show the kinds of program which can be produced by the Microtext program generator. The plug testing program on the left asks the user questions about electrical systems. The program on the right tests your knowledge of the correct procedure when trapped in a burning hotel. Note that both of these programs depend on the computer interacting with the user and then measuring the user's responses against a database of knowledge held in its memory. This sort of application is ideal for program generators

The Last One: For the Apple II and IIe, CP/M-80 and MS-DOS machines
Price: Between £57.50 and £431.25 (inc VAT)
Distributor: D J "AI" Systems Ltd, Summers Orchard, Speke Close, Ilminster, Somerset TA19 9BJ

Microtext: For the Tatung Einstein and CP/M-80 machines
Price: £54.05 (inc VAT)
Distributor: Transdata Ltd, 11 South Street, Havant, Hants

The Quill: For the Spectrum, Commodore 64, Amstrad 464 and Oric
Price: £14.95 (Spectrum, Commodore 64); £16.95 (others)
Distributor: Gilsoft, 30 Hawthorne Road, Barry, South Glamorgan CF6 8LE

Sycero: For MS-DOS machines
Price: £684.25 (inc VAT)
Distrbutor: System C Ltd, 7 Mill Street, Maidstone, Kent ME15 6XW

Certainly, if you boot it up with a kind of BASIC other than what it is looking for, it will correctly identify the DOS disk in any error message

Though superficially similar (in that they are both program generators), Sycero and The Last One are actually quite different in conception, scope and the way they approach the problem of clarifying the user's needs in order to produce usable (and transportable) code. They are also similar in that they both generate programs in BASIC and that they are particularly well-suited for the input, processing, storage, retrieval and printing out of data. They are both different from other programs that perform sophisticated database functions because the resulting code will run independently of the generating system.

In one sense, TLO is geared to the needs of today's users, because it makes the (probably correct) assumption that you will prefer to do all your thinking on screen, doing away with notepad calculations and preparation. But using Sycero and TLO in this manner does not make full use of their potential. The Sycero seven-point plan of action stresses the importance of preparatory planning:

- Plan
- Specify system
- Draw screens
- Check data
- Define program
- Produce print-out
- Generate program

And both Sycero and TLO stress the point quite clearly in their manuals: organisation and preparatory planning are essential to good programming. First, Sycero:

> 'Correcting mistakes due to bad design can take longer than writing the system in the first place. It is always tempting to spend five minutes thinking about a problem and then, Sycero disks in hand, rush off to your computer and make up your system as you go along. This is not a good approach.'
>
> '. . . you should always start by making a comprehensive list of everything you want the computer to produce for you, in terms of what you want to see on the screen ('on-line' information) and what you want in the way of hard copy print-outs ('reports'). Only when you have established what you want out of the system should you decide what information you need to produce the output.'

TLO is less specific but equally to the point:

> 'Plan out the over-all flow of your program and anticipate the mistakes that the end-user is going to make. Then prepare to catch these safely. In fact, design your program as thoroughly as you would plan any other task. As a general rule, it is easier to start out by writing a number of short programs, linked together with a simple menu which calls each of them up in turn when they are needed — later on you will find that this modular approach makes it possible to plan and create large and complex programs with ease.'

Neither of these packages will generate just any kind of program. They won't produce any kind of game, of course — not even a text-only adventure — nor will they generate a word processor or spreadsheet. Their strengths lie in enabling you to manipulate virtually any kind of data, perform calculations upon it (which could be anything from a simple extraction of the VAT element from a gross sum, to a long and complex engineering equation), store it, retrieve it and print it. In this respect, they are both exceedingly powerful.

It is arguable as to whether they work as well as a straight database management program like dBase II (see pages 1152 and 1186), but it is undeniable that they are easier and friendlier to use than such a program, in which its unfriendliness is often directly proportional to its power. How well Sycero and TLO actually work, and how easy they are to use, is something we will examine in the next instalment.

ENTER THE LISTS

Based around lists that can represent either data or functions, LISP allows the programmer to adapt the language to almost every application. We begin here a series on LISP, outlining its use of lists and seeing why it has found so much acceptance in the field of artificial intelligence.

LISP has become quite popular in recent years, mainly because of its inclusion in research and development within the field of artificial intelligence. As interest in the language increased, it became apparent that LISP could act as a general-purpose language with a wide variety of applications.

This realisation generated a huge number of LISP implementations with many diverse applications, from operating system and compiler writing through to adventure game writing. One unfortunate consequence of this proliferation of the language is that, as with many other languages, there are now many LISP dialects. Despite this, most LISP implementations are reasonably standard (because of its simple and straight forward structure) and translating LISP from one machine to another is usually quite easy.

One of the problem areas you're most likely to come across when using LISP is the lack of a certain function or instruction. This is mainly due to the lack of any official standard, leaving only those functions the programmers feel to be the most important. However, as we will see, it is extremely easy to extend the language, by adding any new instructions that you may require.

LISP is entirely different from the more common languages, such as PASCAL, FORTRAN and BASIC. It has a unique and very simple and uniform syntax structure. This structure makes LISP ideal for data manipulation and matching problems.

The inherent list structure is easily adaptable to all the major computer information structures, allowing the language to be used for searching and sorting, arithmetic functions and even game playing. In addition, most microcomputers support special instructions to make use of their own sound and graphics features. This is certainly the case with Acornsoft LISP for the BBC Micro, which we will be using in this series. Don't worry if you have a different version of the language because you'll find that everything is directly transferable.

The basis of LISP, as you might expect, is the *list* data structure — the language's name derives from the term 'list processing'. A list is, in some ways, similar to the more familiar 'array' found on most micros. However, unlike the array, a list does not have any specific length.

Any list of items may be shown by enclosing it in parentheses, in the form:

```
(a b c d e . . .)
```

where a, b, c and so on are the list's elements. The technical term for these is *atoms*, and they may be numeric data, non-numeric (character data or variables) or even another list. Note that we do not separate elements of a list from each other by using commas — we use spaces instead.

Operations are performed on lists using functions, in much the same way as a BASIC function (DEF FN command) performs an operation on its arguments to produce a result. A function is written in the form:

```
(func a b c d . . .)
```

where func is the function name and a, b, c and so on are its arguments. As you can see, the function looks very much like the first element of the list.

So, for example, a list consisting of the first six prime numbers would be written as:

```
(1 2 3 5 7 11)
```

If we wanted to add these together in BASIC, we would use:

```
1 + 2 + 3 + 5 + 7 + 11
```

In LISP, we would use the plus function and write:

```
(plus 1 2 3 5 7 11)
```

which would return the answer 29. One of the features of the PLUS function is that it can take any number of arguments; so we could just as easily write:

```
(PLUS 1 2 3 (PLUS 5 7 11))
```

Here, the innermost PLUS function is evaluated first and returns the answer 23. This is then used as the fourth argument for the outermost PLUS to return the final answer of 29. An important point here is the ease with which functions may be nested. By nesting another function, SETQ, we can assign the result to the variable A:

Lisp Implementations

Despite the increasing interest shown in LISP, there are currently few affordable versions of the language for micro users. Acornsoft LISP is available for the BBC Micro and Acorn Electron for under £20, though most users will find that they have to buy the LISP User Guide, which costs a further £7.50.

Other micros are less well catered for. There are, however, a number of CP/M versions of the language — notably Toolworks Lisp/80 (at around £50) and iLISP (around £80). Another CP/M version, Microsoft's muLISP-83, is considerably more expensive, retailing at about £200.

The CP/M-based interpreters mentioned above should run on most home micros supporting CP/M (Memotech, Einstein, etc), although Amstrad owners will not be able to use them due to lack of memory space (the minimum required is 48K). Toolworks is probably the best value, with a storage space of approximately 3,600 list cells and 11,000 atom name characters in 48K. All the programs may be purchased from Grey Matter Ltd, 4 Prigg Meadow, Ashburton, Devon, TQ13 7DF

```
(SETQ A (PLUS 1 2 3 5 7 11))
```

where SETQ has two arguments — the variable A and a function that evaluates to the integer 29. Of course, SETQ is itself a function and so must return a result (in this case the value 29). So, the following assignment, which is illegal in most BASICS:

```
LET B = 1 + 2 + 3 + 5 + 7 + 11
```

could be written in LISP as:

```
(SETQ B (SETQ A (PLUS 1 2 3 5 7 11)))
```

By introducing the TIMES function, we can assign, say, two times the value of A to the variable B:

```
(SETQ B (TIMES 2 (SETQ A (PLUS 1 2 3 5 7 11))))
```

This would first of all evaluate PLUS to return the value 29, which would be assigned to the variable A by the innermost SETQ function. This function would, in turn, give the result 29, to be used as the second argument of the TIMES function. The new result of 58 would then be passed to the outermost SETQ function to place the value into the variable B. The entire expression would give the result 58, which could then be used for further functions, and so on.

Notice that the brackets were becoming quite prolific in the last example. This is a feature of LISP, and keeping track of all these brackets can be quite tedious, although if the program is well designed, they tend to take care of themselves. In addition, some LISP systems help out by informing you of the number of outstanding brackets that you have. For example, when using Acornsoft LISP on the BBC Micro, the number of arrows given as the start of line prompt indicates the number of brackets that need closing to complete the expression.

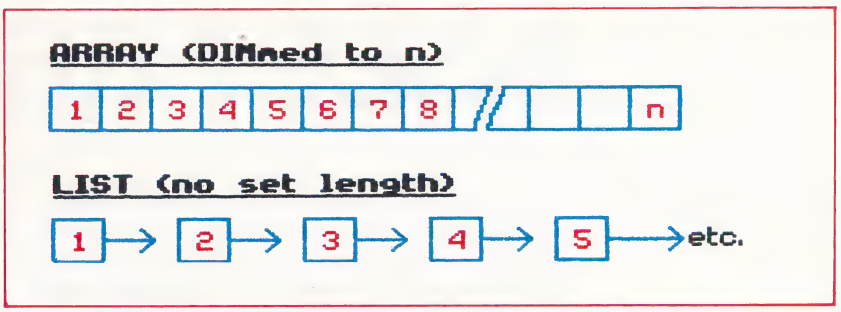
In the last expression we introduced the function TIMES, and gave it two arguments: the integer value 2 and a list expression that had previously been seen to give the answer 29. However, TIMES, like PLUS, may have a variable number or arguments. Thus all the following would be legal expressions:

```
(TIMES 1 2 4 8 16)
(TIMES 1 2 4 8 (TIMES 4 4))
(TIMES 1 2 (TIMES 2 2) (TIMES 2 4) (PLUS 8 8))
```

and would all return the result 1024.

In reality, most LISP implementations have a limit on the number of arguments that these types of functions can have. For instance, Acornsoft LISP has a limit of 28 arguments, and some other versions have even more stringent restrictions.

By now we can see that an instruction in LISP is merely a list of items, in which the first element is the function to be performed, and the subsequent items are the function's arguments. These, in turn, may themselves be lists with the first argument of each being a function that returns a result. The question now arises of what to do if we don't want a function. It's quite possible that we merely want to set up a list of data items, say, to use as a title. We can't write:




```
(HOME COMPUTER ADVANCED COURSE)
```

as a list of four data items, because LISP would try to evaluate HOME as a function name, with three arguments, and would think that each of the arguments (COMPUTER, ADVANCED and COURSE) were three variable names. We can tell LISP not to evaluate an expression by preceding it with a single quote ('), so we would write the above as:

```
'(HOME COMPUTER ADVANCED COURSE)
```

We could then assign this list to a variable:

```
(SETQ MAG '(HOME COMPUTER ADVANCED COURSE))
```

which would assign a list of four non-numeric items to the variable MAG.

Note that there is no variable type in LISP. If we were using BASIC, an integer variable would be of the form A%, a real variable A and a string variable A$. In LISP, we don't distinguish between variables in this way, so all the following are legal expressions:

```
(SETQ A 3)              A = the integer 3.
(SETQ A 'COMPUTER)      A = the string 'COMPUTER'.
(SETQ A (PLUS 2 4 8))   A = the result of 2+4+8.
(SETQ A '(1 2 4 8))     A = the list (1 2 4 8).
(SETQ A '(HOME COMPUTER)) A = the list (HOME COMPUTER).
(SETQ A B)              A = the value of variable B.
```

Though integer and string types are fairly standard, few microcomputers support a version of LISP that performs floating-point arithmetic. In most cases, integer arithmetic is sufficient, and in the unlikely event that floating-point arithmetic is needed, it could no doubt be simulated using the standard integers.

At this stage, it might seem difficult to realise how the language could be useful. In the next instalment, we will look at LISP functions that can manipulate their data arguments, and see how the principles of recursion are used to make LISP an extremely powerful language.

Lists Vs Arrays

Lists have two main advantages over arrays. First, memory does not need to be reserved for them before it is needed; second, a list is a 'dynamic' structure, which is to say that it need have no fixed length and can shrink or expand to accommodate data during program execution. Furthermore, lists lend themselves more easily to the recursive procedures frequently encountered in AI programming

INSIDE INTERRUPTS

The Z80 CPU offers the machine code programmer three maskable interrupt modes, which allow you to 'borrow' processor time when required. We examine the role of interrupts within the Spectrum's OS, and see how they can be vectored to point to your own machine code routines.

The interrupt-driven facilities of the Spectrum's operating system are less straightforward than those used on the BBC Micro (see page 1077). Great care should be taken when using interrupts on the Spectrum. Should they be inadvertently disabled, or 'turned off', the functioning of the machine can be seriously affected — the keyboard may not be read, for example, or the system could crash during the execution of BASIC programs.

The Z80 CPU at the heart of the Spectrum responds to two different types of interrupt — maskable and non-maskable (NMI) interrupts. The difference between these is quite simple. We can program the CPU to ignore a maskable interrupt signal, but the processor will always respond to a non-maskable interrupt.

NMI interrupts are a little impractical to use on the Spectrum, because the NMI-handling ROM routine appears to be bugged. The original intention of the designers of the OS was to allow the user to specify an address in the unused system variables at locations &5CB0 and &5CB1, and this was to be jumped to whenever the CPU received an NMI pulse. However, the usual response of the Z80 is to execute the machine code beginning at address &66, which generally results in a complete reset of the system. Consequently, we will concentrate our attention on the use of maskable interrupts.

There are several ways in which the Z80 CPU can respond to a maskable interrupt, and these alternative responses are termed the *modes* of interrupt operation. Here, we will deal with only those modes that are of significance to the Spectrum operating system.

On initialisation — which occurs when you turn the machine on or when you issue a NEW command — the CPU sets up interrupt mode 1, or IM 1. The Spectrum normally runs in this interrupt mode. The interrupt pulses are fed to the CPU by the Sinclair ULA (Uncommitted Logic Array) at a rate of 50 interrupts every second. In IM 1 mode, the CPU executes a RST &0038 instruction upon receiving an interrupt signal. This results in a jump to routines that read the keyboard and update the FRAMES counter, located at RAM locations 23672 to 23674. (These three bytes form a 24-bit counter that is hence updated every 20 milliseconds.)

After a line of BASIC has been interpreted, the BASIC interpreter waits for an interrupt before it proceeds to the next statement interpretation. This means that if you disable the interrupts, BASIC program execution will come to a halt.

Maskable interrupts will be ignored by the CPU after a DI (Disable Interrupt) instruction has been executed, and they are re-enabled by an EI (Enable Interrupt) command.

Various routines within the Spectrum operating system require interrupts to be disabled while they are being executed. These are generally time-dependent routines such as the BEEP routine of the sound generator and the tape save and load routines. Interrupts can also be temporarily disabled by the ZX printer, Interface 1 or the Microdrive. Obviously, the OS re-enables interrupts once it has finished with the routine.

One practical result of various routines disabling interrupts is that while these routines are being executed the FRAMES counter is not incremented. This will cause it to 'lose time' while these operations are being performed. As an interrupt can occur while the CPU is running one of your programs, it's a good idea to disable the interrupts for any portion of your program in which accurate timing is important. However, it is

Anticipating All Possibilities

Although it can be assumed for most purposes that the data bus on the Spectrum will hold 255 when an interrupt is generated, certain peripherals (such as the Kempston joystick interface) can alter this value. The easiest way to get around this problem is to ensure that you do not use interrupt mode 2 (IM2) with peripherals attached.

An alternative method, which will allow for all possible values on the data bus, is to fill a page of memory with identical values. The page number is then loaded into the I register before selecting IM2. When an interrupt occurs, the Z80 will then fetch an interrupt service routine address from a position within the specified page — the exact position to be determined by the value on the data bus.

First, we load the I register with the value &FC, and fill bytes &FC00 to &FD00 with the value &FB. We then select IM2. If the data bus holds &C3 when the next interrupt occurs, the Z80 will then pick up the interrupt service routine address from location &FCC3 — the &FC part of the address having been supplied by the I register. The address picked up will be, of course, &FBFB — since all 256 bytes of page FC (plus byte zero of page &FD) have been set to the same value.

There are two significant points to note here. The first is that your interrupt service routine (ISR) will always have to be at an address with identical lo- and hi-bytes (for example: &C4C4 or &FDFD). The second point is that you must remember to allow for the fact that the data bus may actually hold &FF when an interrupt occurs. In this case the Z80 will look for the ISR routine at addresses nnFF (lo-byte) and (nn+1)00 (hi-byte), where nn is the value of the I register. For this reason, you must remember to set the first byte of the following page

vital to re-enable them before you make the return to BASIC!

VECTORED INTERRUPTS

To make any practical use of Spectrum interrupts, it is necessary to change the interrupt mode to one that allows more versatility. The most useful interrupt mode is IM2, which is more complicated in action than interrupt mode 1. Whereas IM1 always results in a jump to address &0038 (using the RST &0038 instruction), IM2 can be used to jump to a routine elsewhere in memory. The address that the CPU jumps to is specified using what is known as an *interrupt vector.*

In a vectored interrupt, the vector holds the address of the *interrupt service routine* that is to be executed when a maskable interrupt occurs. The CPU knows where the vector is located in memory by making use of a special register in the Z80 called the *I register.* The address of the interrupt vector routine is found by combining the contents of the I register with those of the data bus at the instant the interrupt occurs. In some systems the device causing an interrupt will put a byte on the data bus to tell the CPU what caused the interrupt.

The Sinclair ULA does not, however, put a value on the data bus when it sends an interrupt signal to the Z80, but the way in which the Spectrum hardware is arranged ensures that the data bus holds the value 255 when no other input is applied to it. The I register, therefore, provides the hi-byte of the 16-bit address of the interrupt vector, and the value on the data bus (in this case &FF) provides the lower byte of the address. Thus, the interrupt vector will always be located at a page boundary in memory, with the lo-byte of the interrupt service routine held at address &nnFF and the hi-byte at address &(nn+1)00 (where nn is the contents of the I register).

For example, if the register holds the value &FB, the address of the vector will be at &FBFF. The lo-byte of the vector, at location &FBFF, will hold the lo-byte of the address of the interrupt service routine, and the hi-byte of the vector, at address &FC00, will hold the hi-byte of the ISR address.

There are restrictions on where the ISR can be located in memory — the first 16 Kbytes, for example, is assigned to the ROM, and therefore cannot be used. Secondly, hardware problems prevent the correct operation of the OS with an I register value in the range 64 to 127.

USING INTERRUPT MODE 2

Let's now look at how the Z80 is put into interrupt mode 2, and set up the interrupt vector with the address of the ISR to be used:

```
F3          di
210000      ld   hl,ADDRESS      ;get ISR address in HL
22FFFB      ld   (#FBFF),hl      ;get address into vector
3EFB        ld   a,#FB           ;hi-byte of vector...
ED47        ld   i,a             ;...into I register
ED5E        IM   2               ;set int mode 2
FB          ei                   ;re-enable interrupts
C9          ret                  ;back to BASIC
```

Of course, this listing assumes that there is a routine at ADDRESS to handle the interrupts — if there isn't, a crash is likely to occur. Furthermore, any interrupt service routine should ideally perform all of the tasks normally carried out by the Spectrum in its usual interrupt mode. This is best done with a call to the routine at address &38. The following Assembly language program changes the interrupt mode to IM2 and causes the CPU to execute the routine at ADDRESS on each interrupt (which in this case simply performs the usual functions that the Spectrum carries out in its normal interrupt mode).

```
            org  60000          ;specify start add
     vector: equ  #FEFF         ;FEFF is vector add
216FEA change: ld   hl,addres   ;get address into HL
22FFFE      ld   (vector),hl    ;set up vector
F3          di                  ;disable interrupts
3EFE        ld   a,#FE          ;set up the...
ED47        ld   i,a            ;...I register
ED5E        im   2              ;change int mode
FB          ei                  ;re-enable interrupts
C9          ret                 ;back to BASIC
F3     addres: di                 ;turn off interrupts
FF          rst  #38            ;normal IM1 procedure
FB          ei                  ;re-enable interrupts
C9          ret
```

Calling the routine at address CHANGE will set up the new interrupt mode and the vector. Once this is done, the routine at ADDRESS will be executed every one-fiftieth of a second. At the moment this routine doesn't do anything particularly useful, but we'll give it some handy functions shortly.

If it's necessary to change the interrupt mode back to normal at any time during the program, a routine such as the one following can be used to peform this task:

```
F3          di
3E3F        ld   a,#3f          ;reset I register...
ED47        ld   i,a            ;...to its usual value
ED56        im   1              ;normal mode
FB          ei                  ;re-enable interrupts
C9          ret
```

To make the interrupt service routine run some code of our own, we simply treat the code as a subroutine and CALL it in the way shown here:

```
F3     addres: di
F5             push af          ;save registers...
C5             push bc
D5             push de
E5             push hl
FF             rst  #38         ;call normal ISR
F3             di               ;ISR did EI, so DI again
CD0000         call PROG_ADD    ;call our own routine
E1             pop  hl          ;restore registers...
D1             pop  de
C1             pop  bc
F1             pop  af
FB             ei
C9             ret
```

Obviously, if your interrupt service routine is lengthy, it will slow the Spectrum down slightly, which will affect program execution and the rate at which FRAMES is incremented. Finally, always remember to disable interrupts during your routine so as to prevent a second interrupt occurring halfway through the servicing of the first one.

Interrupt Music Program

The following program shows how interrupts can be used to provide 'background music' on your Spectrum.

The Note Data Compiler is a BASIC program that enables you to write your own tunes, whereas the Assembly listing (or the BASIC Loader if you don't have an assembler) will enable you to set up the interrupt-driven music routine and pass to it the data compiled by the Note Data Compiler.

RUNning the Note Data Compiler presents a menu with three options. 'C – COMPILE' will, if selected, change the tune (held in the data statements between lines 10 and 900) into a block of code that can be used by the interrupt routine. 'P' will play the tune so you can decide what changes (if any) you wish to make. 'R' will return you to BASIC so that you can SAVE the compiled data or change the music data in the data statements.

Note that if you use the 'C' option, you will be asked to specify where in memory you wish to store the note data. Make sure that you store it above RAMTOP, the value of which you will have had to alter previously to make room for the data. When the notes have been compiled and stored, the base address and length of the code will be displayed. You should make a note of these because you will need this information later. To save code, return to BASIC and enter: SAVE "NOTEDATA" CODE (base address), (no of bytes long). Finally, if you wish to use your own tunes, just get rid of lines 10 to 90 and put your own data anywhere between lines 10 and 900.

Type in the Assembly listing using an assembler and save it to tape. If you do not have an assembler, use the BASIC Loader.

The routine has a base address of 65021, so before loading the code into memory ensure that you have lowered RAMTOP accordingly (CLEAR 65000 will do). Having assembled and loaded the code, you will need to load the code compiled by the Note Data Compiler (below). Remember to lower RAMTOP again to allow for the note data.

The interrupt music program needs to know the exact location of the note data to be played before it can operate. You can pass this information to the routine using the following short BASIC program:

```
10 LET L=65152: LET
   V=(*base address of note
   data*)
20 POKE L+1,INT(V/256):
   POKE L,V-PEEK (L+1)*256
```

Having done that, RAND USR 65041 will set up IM2 and start playing your tune. RAND USR 65071 will re-select IM1 and turn the music off

Note Data Compiler

```
  1 REM >>> INTERRUPT MUSIC <<<
  2 REM >>Note Data  Compiler<<
  3 REM
  6 REM *NOTE DATA IN STANDARD SPECTRUM BEEP FORM
AT *
  7 REM
  9 RESTORE
 10 DATA 1,12,2,9,1,9,1,9,1,8,1,9,2,17,1,12,2,12,
1,9,2,10,1,10,2,10,1,12,5,14
 20 DATA 1,14,2,7,1,7,1,7,1,6,1,7,2,16,1,14,2,14,
1,10,2,9,1,9,2,9,1,10,5,12
 30 DATA 1,12,2,9,1,9,1,9,1,8,1,9,2,17,1,12,2,12,
1,12,2,11,1,19,2,19,1,19,5,19
 40 DATA 1,17,2,16,1,19,1,19,1,18,1,19,2,14,1,19,
1,19,1,18,1,19,2,12,1,11,2,12,1,11,3,12
 50 DATA 3,10,1,9,1,8,1,9,2,14,1,12,3.2,9,3.2,5,3
.2,2,3.2,7,5,5
 60 DATA 1,5,1,7,1,9,1,10,2,16,1,14,3.2,12,3.2,17
,3.2,16,3.2,14,5,12
 70 DATA 1,12,2,14,1,14,1,14,1,13,1,14,3,16,3,9
 80 DATA 2,17,1,17,1,19,1,17,1,19,5,21
 90 DATA 1,21,2,19,1,17,2,14,1,10,3.2,9,3.2,5,3.2
,7,3.2,4,5,5
997 REM
998 REM *DO NOT ERASE LINE 999*
999 DATA 255,255
1000 REM
2000 REM **** MENU ****
2010 BORDER 1: PAPER 1: INK 6
2020 CLS
2030 PRINT AT 5,4;"INTERRUPT MUSIC COMPILER";AT 10
,5;"C -- Compile note data";AT 12,5;"P -- Play tun
e in beeps";AT 14,5;"R -- Return to BASIC"
2040 LET a$=INKEY$
2050 IF a$="c" OR a$="C" THEN  GO TO 5000
2060 IF a$="p" OR a$="P" THEN  GO TO 3000
2070 IF a$="r" OR a$="R" THEN  CLS : STOP
2080 GO TO 2040
3000 RESTORE
3010 READ d,f: IF d=255 AND f=255 THEN  GO TO 2000
3020 BEEP d/10,f-6: GO TO 3010
5000 RESTORE : CLS : INPUT "BASE ADDRESS OF NOTE D
ATA? ";dd1: POKE 23301,INT (dd1/256): POKE 23300,d
d1-(256*PEEK 23301): CLEAR dd1-1: LET dd1=PEEK 233
00+256*PEEK 23301
5005 LET f=0: LET d=dd1
5010 READ delay,pitch
5015 LET delay=INT (delay*6)
5020 LET freq=(1.0594631^pitch)*256
5030 LET bip=INT ((437500/freq)-30.125)
5035 IF delay=255*6 THEN  POKE d,255: GO TO 6000
5037 LET f=f+1: PRINT AT 10,10;"NOTE )";f
5040 POKE d,delay
5060 POKE d+1,bip-(INT (bip/256)*256)
5070 POKE d+2,INT (bip/256)
5080 LET d=d+3
5090 GO TO 5010
6000 PAUSE 50: CLS : LET len=(d+2)-dd1
6010 PRINT "LOCATION OF NOTE DATA IN MEMORY=";dd1'
''"NUMBER OF BYTES OF NOTE DATA....";len
6020 INPUT "PRESS 'ENTER' TO RETURN TO MENU "; LIN
E a$: GO TO 2000
```

BASIC Loader

```
 10 RESTORE 9000
 20 CLEAR 64000
 30 LET cqs=0: FOR f=65021 TO 65156
 40 READ dat: POKE f,dat
 50 LET cqs=cqs+dat
 60 NEXT f
 70 IF cqs<>18850 THEN  PRINT "ERROR IN DATA !!!"
: STOP
 80 PRINT "OK NO PROBLEMS IN DATA"
 90 STOP
9000 DATA 0,0,221,229,229,197,213,245,195,52,254,2
41,209,193,225,221,225,195,56,0,33,255,253,34,253,
253,243,62,253,237,71,237,94,42,128,254,34,126,254
,62,1,50,130,254,61,50,131,254,251,201,243,237,86,
251,201,58,130,254,254,0,202,8,254,58,131,254,254,
0,202,75,254,61,50,131,254,195,8,254,42,126,254,12
6,254,255,202,108,254,50,131,254,35,126,35,95,126,
35,34,126,254,103,107,17,4,0,205,181,3,243,195,8,2
54,42,128,254,34,126,254,62,1,50,130,254,61,50,131
,254,195,8,254,0,0,0,0,0,0,
```

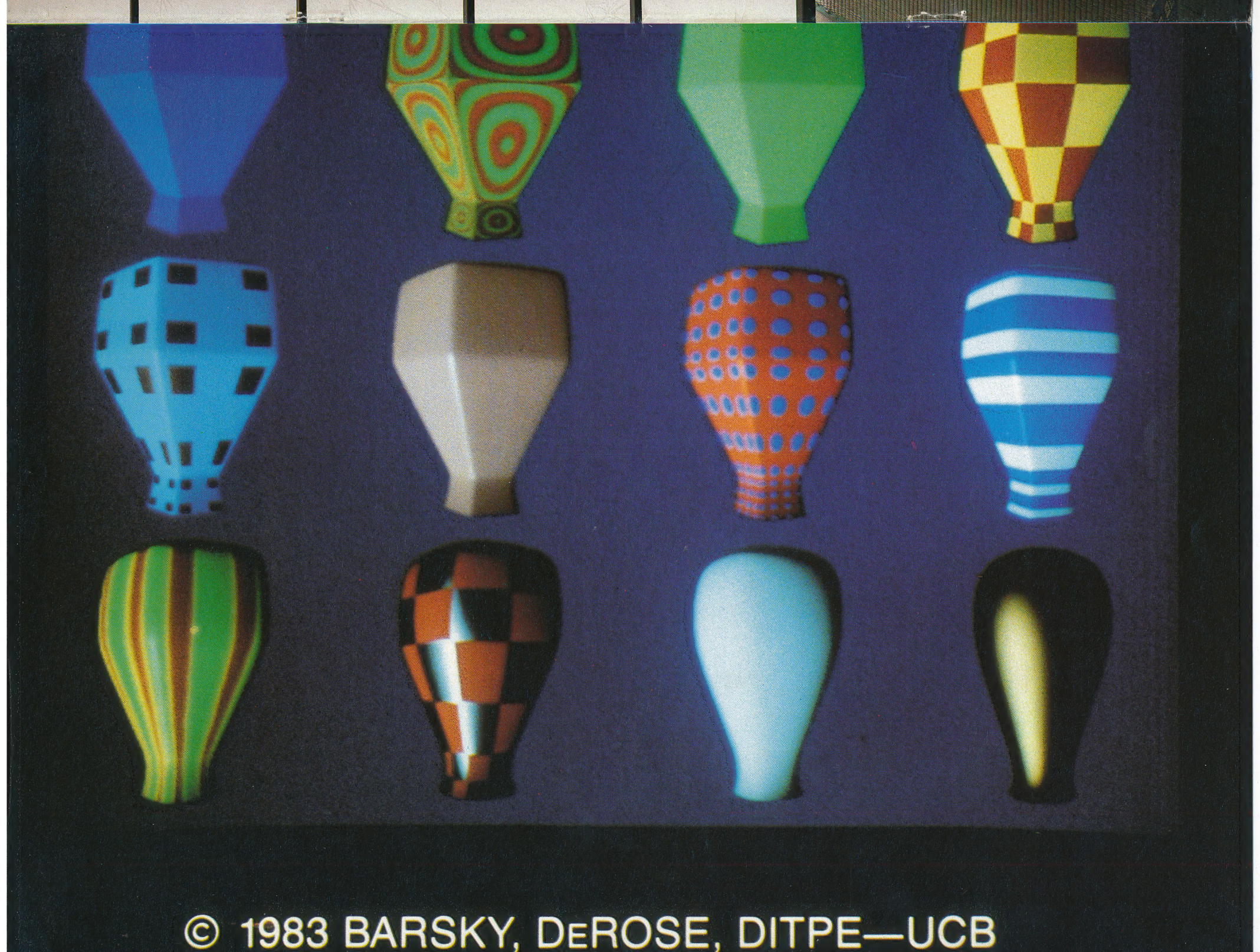
Assembly Listing

```
                org  65021
0000    vect:   defw 0                  ;2 bytes for vector
DDE5    inter:  push ix
E5              push hl
C5              push bc
D5              push de
F5              push af
C334FE          jp   start              ;jp to music routine
F1      enprg:  pop  af
D1              pop  de
C1              pop  bc
E1              pop  hl
DDE1            pop  ix
C33800          jp   56                 ;jp to ROM ISR
21FFFF  on:     ld   hl,inter           ;set up vector
22FDFD          ld   (vect),hl
F3              di
3EFD            ld   a,253
ED47            ld   i,a                ;get vect hi-byte in I
ED5E            im   2                  ;set int mode 2
2A80FE  initi:  ld   hl,(datad)         ;point to data
227EFE          ld   (locat),hl
3E01            ld   a,1                ;switch = 1 = play note
3282FE          ld   (swtch),a
3D              dec  a                  ;delay = 0
3283FE          ld   (delay),a
FB              ei
C9              ret                     ;back to BASIC
F3      off:    di
ED56            im   1                  ;set int mode 1
FB              ei
                ret
C9      ;
3A82FE  start:  ld   a,(swtch)          ;get switch in a
FE00            cp   0
CA08FE          jp   z,enprg            ;if swtch 0 then end
3A83FE          ld   a,(delay)
FE00            cp   0
CA4BFE          jp   z,nwnot            ;if delay 0 then play
3D              dec  a                  ;delay=delay-1
3283FE          ld   (delay),a          ;store new delay value
C308FE          jp   enprg              ;goto end routine
2A7EFE nwnot:   ld   hl,(locat)         ;get data location
7E              ld   a,(hl)             ;get next delay in a
FEFF            cp   255                ;255 = data finished
CA6CFE          jp   z,reset            ;data ended so reset
3283FE          ld   (delay),a          ;store new delay
23              inc  hl                 ;point to next databyte
7E              ld   a,(hl)             ;get it in a
23              inc  hl                 ;point to next databyte
5F              ld   e,a                ;store byte1 in e
7E              ld   a,(hl)             ;store byte2 in a
23              inc  hl                 ;point to next byte
227EFE          ld   (locat),hl         ;store next-byte-address
67              ld   h,a                ;get frequency data...
6B              ld   l,e                ;...in HL
110400          ld   de,4               ;note duration of 4
CDB503          call 949                ;call ROM beep routine
F3              di                      ;ROM-BEEP did EI, so DI
C308FE          jp   enprg              ;goto end routine
2A80FE reset:   ld   hl,(datad)         ;reset various...
227EFE          ld   (locat),hl         ;...pointers
3E01            ld   a,1
3282FE          ld   (swtch),a
3D              dec  a
3283FE          ld   (delay),a
C308FE          jp   enprg              ;goto end routine
0000    locat:  defw 0                  ;reserve memory for...
0000    datad:  defw 0                  ;various pointers...
00      swtch:  defb 0
00      delay:  defb 0
```

DATABASE

Here, courtesy of Commodore Business Machines, we publish another instalment of the Commodore 64's memory map

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
MEMSTR	0281–0282	641–642	Pointer: Bottom of Memory for O.S.
MEMSIZ	0283–0284	643–644	Pointer: Top of Memory for O.S.
TIMOUT	0285	645	Flag: Kernal Variable for IEEE Timeout
COLOR	0286	646	Current Character Color Code
GDCOL	0287	647	Background Color Under Cursor
HIBASE	0288	648	Top of Screen Memory (Page)
XMAX	0289	649	Size of Keyboard Buffer
RPTFLG	028A	650	Flag: REPEAT Key Used, $80 = Repeat
KOUNT	028B	651	Repeat Speed Counter
DELAY	028C	652	Repeat Delay Counter
SHFLAG	028D	653	Flag: Keyb'rd SHIFT Key/ CTRL Key/C= Key
LSTSHF	028E	654	Last Keyboard Shift Pattern
KEYLOG	028F–0290	655–656	Vector: Keyboard Table Setup
MODE	0291	657	Flag: $00=Disable SHIFT Keys, $80 = Enable SHIFT Keys
AUTODN	0292	658	Flag: Auto Scroll Down, 0 = ON
M51CTR	0293	659	RS-232: 6551 Control Register Image
M51CDR	0294	660	RS-232: 6551 Command Register Image
M51AJB	0295–0296	661–662	RS-232 Non-Standard BPS (Time/2-100) USA
RSSTAT	0297	663	RS-232: 6551 Status Register Image
BITNUM	0298	664	RS-232 Number of Bits Left to Send



© 1983 BARSKY, DEROSE, DITPE—UCB